USENIX

SAN ANTONIO CONFERENCE PROCEEDINGS

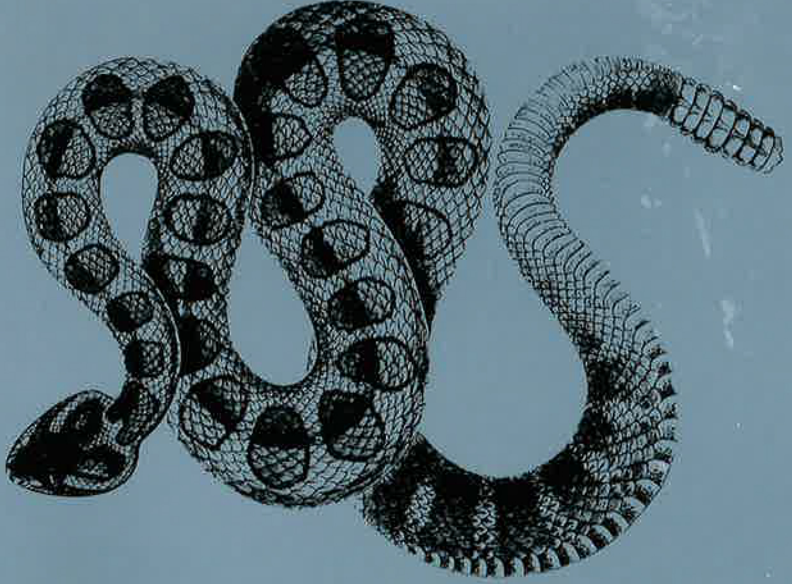
SUMMER

1992

CONFERENCE PROCEEDINGS

USENIX

SUMMER 1992 TECHNICAL CONFERENCE
SAN ANTONIO, TEXAS • JUNE 8–12, 1992



USENIX, THE UNIX AND ADVANCED COMPUTING SYSTEMS PROFESSIONAL AND TECHNICAL ASSOCIATION

USENIX Association

Proceedings of the Summer 1992 USENIX Conference

June 8 – June 12, 1992
San Antonio, Texas, USA

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $23 for members and $30 for nonmembers.

Outside the U.S.A and Canada, please add
$14 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1992 Winter	San Francisco		1987 Winter	Washington, DC
1991 Summer	Nashville		1986 Summer	Atlanta
1991 Winter	Dallas		1986 Winter	Denver
1990 Summer	Anaheim		1985 Summer	Portland
1990 Winter	Washington, DC		1985 Winter	Dallas
1989 Summer	Baltimore		1984 Summer	Salt Lake City
1989 Winter	San Diego		1984 Winter	Washington, DC
1988 Summer	San Francisco		1983 Summer	Toronto
1988 Winter	Dallas		1983 Winter	San Diego
1987 Summer	Phoenix			

ISBN 1-880446-44-8

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

TABLE OF CONTENTS

CPP: Threat or Menace?

ACKNOWLEDGMENTS

PROGRAM CHAIR
Rick Adams, *UUNET Technologies, Inc.*

PROGRAM COMMITTEE
Rick Adams, *UUNET Technologies, Inc.*
Vadim Antonov, *DEMOS*
Keith Bostic, *CSRG, University of California, Berkeley*
Bill Cheswick, *AT&T Bell Laboratories*
Judith E. Grass, *AT&T Bell Laboratories*
Carl S. Gutekunst, *Pyramid Technology Corporation*
Guy Harris, *Auspex Systems, Inc.*
Kenneth Ingham, *Kenneth Ingham Consulting*
Rob Kolstad, *Berkeley Software Design, Inc.*
Piers Lauder, *University of Sydney*
A. Elein Mustain, *Ingres, an ASK Copmany*
David Nichols, *Xerox PARC*
Margo Seltzer, *University of California, Berkeley*
Jim Thompson, *Smallworks, Inc.*

PROGRAM COMMITEE SCRIBE
Andrew S. Partan, *UUNET Technologies, Inc.*

PROCEEDINGS PRODUCTION
Carolyn Carr, *USENIX Association*
Malloy Lithographing, Inc.
Rob Kolstad, *Berkeley Software Design, Inc.*

INVITED TALKS COORDINATORS
Tom Cargill, *Consultant*
Andrew Hume, *AT&T Bell Laboratories*

BOF COORDINATOR
Gene Cristofor, *AT&T Bell Laboratories*

TUTORIAL COORDINATOR
Daniel V. Klein, *USENIX Association*

WORK-IN-PROGRESS COORDINATOR
Lisa A. Bloch, *Sun User Group*

TERMINAL ROOM COORDINATOR
Eve Podet, *mt Xinu*

USENIX MEETING PLANNER
Judith F. Desharnais, *USENIX Association*

AUTHOR INDEX

Implementing Lightweight Threads

D. Stein, D. Shah – SunSoft, Inc.

ABSTRACT

We describe an implementation of a threads library that provides extremely lightweight threads within a single UNIX process while allowing fully concurrent access to system resources. The threads are lightweight enough so that they can be created quickly, there can be thousands present, and synchronization can be accomplished rapidly. These goals are achieved by providing user threads which multiplex on a pool of kernel-supported threads of control. This pool is managed by the library and will automatically grow or shrink as required to ensure that the process will make progress while not using an excessive amount of kernel resources. The programmer can also tune the relationship between threads and kernel supported threads of control. This paper focuses on scheduling and synchronizing user threads, and their interaction with UNIX signals in a multiplexing threads library.

Introduction

This paper describes a threads library implementation that provides extremely lightweight threads within a single UNIX process while allowing fully concurrent access to system resources. The threads are lightweight enough so that they can be created quickly, there can be thousands present, and synchronization can be accomplished rapidly. These goals are achieved by providing user threads which multiplex on a pool of kernel-supported threads of control. The implementation consists of a threads library and a kernel that has been modified to support multiple threads of control within a single UNIX process [Eykholt 1992].

The paper contains seven sections: The first section is an overview of the threads model exported by the library. The second section is an overview of the threads library architecture. Readers familiar with the SunOS Multi-thread Architecture [Powell 1991] can skip the first two sections. The third section details the interfaces supplied by the kernel to support multiple threads of control within a single process. The fourth section details how the threads library schedules threads on kernel-supported threads of control. The fifth section details how the library implements thread synchronization. The sixth section details how the threads library implements signals even though threads are not known to the kernel. The final section briefly describes a way to implement a debugger that understands threads.

Threads Model

A traditional UNIX process has a single thread of control. In the SunOS Multi-thread Architecture [Powell 1991], there can be more than one thread of control, or simply more threads, that execute independently within a process. In general, the number or identities of threads that an application process applies to a problem are invisible from outside the process. Threads can be viewed as execution resources that may be applied to solving the problem at hand.

Threads share the process instructions and most of its data. A change in shared data by one thread can be seen by the other threads in the process. Threads also share most of the operating system state of a process. For example, if one thread opens a file, another thread can read it. There is no system-enforced protection between threads.

There are a variety of synchronization facilities to allow threads to cooperate in accessing shared data. The synchronization facilities include mutual exclusion (mutex) locks, condition variables, semaphores and multiple readers, single writer (readers/writer) locks. The synchronization variables are allocated by the application in ordinary memory. Threads in different processes can also synchronize with each other via synchronization variables placed in shared memory or mapped files, even though the threads in different processes are generally invisible to each other. Such synchronization variables must be marked as being process-shared when they are initialized. Synchronization variables may also have different variants that can, for example, have different blocking behavior even though the same synchronization semantic (e.g., mutual exclusion) is maintained.

Each thread has a program counter and a stack to maintain of local variables and return addresses. Each thread can make arbitrary system calls and interact with other processes in the usual ways. Some operations affect all the threads in a process. For example, if one thread calls `exit()`, all threads are destroyed.

Each thread has its own signal mask. This permits a thread to block asynchronously generated signals while it accesses state that is also modified by a signal handler. Signals that are synchronously generated (e.g., SIGSEGV) are sent to the thread that caused them. Signals that are generated externally are sent to one of the threads within a process that has it unmasked. If all threads mask a signal, it is set pending on the process until a thread unmasks that signal. Signals can also be sent to particular threads within the same process. As in single-threaded processes, the number of signals received by the process is less than or equal to the number sent.

All threads within a process share the set of signal handlers[1]. If the handler is set to `SIG_IGN` any received signals are discarded. If the handler is set to `SIG_DFL` the default action (e.g., stop, continue, exit) applies to the process as a whole.

A process can fork in one of two ways: The first way clones the entire process and all of its threads. The second way only reproduces the calling thread in the new process. This last is useful when the process is simply going to call `exec()`.

Threads Library Architecture

Threads are the programmer's interface for multi-threading. Threads are implemented by a dynamically linked library that utilizes underlying kernel-supported threads of control, called lightweight processes (LWPs)[2], as shown in Figure 1.

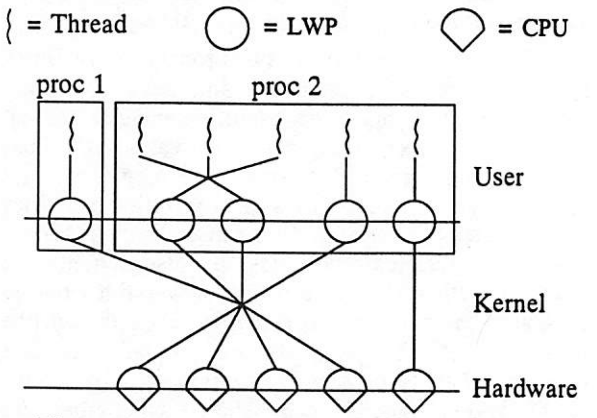


Figure 1: Multi-thread Architecture Examples

Each LWP can be thought of as a virtual CPU which is available for executing code or system calls. Each LWP is separately dispatched by the kernel, may perform independent system calls, incur independent page faults, and may run in parallel on a multiprocessor. All the LWPs in the system are scheduled by the kernel onto the available CPUs according to their scheduling class and priority.

The threads library schedules threads on a pool of LWPs in the process, in much the same way as the kernel schedules LWPs on a pool of processors[3]. Threads themselves are simply data structures and stacks that are maintained by the library and are unknown to the kernel. The thread library can cause an LWP to stop executing its current thread and start executing a new thread without involving the operating system. Similarly, threads may also be created, destroyed, blocked, activated, etc., without operating system involvement. LWPs are relatively much more expensive than threads since each one uses dedicated kernel resources. If the threads library dedicated an LWP to each thread as in [Cooper 1990] then many applications such as data bases or window systems could not use threads freely (e.g. one or more per client) or they would be inefficient. Although the window system may be best expressed as a large number of threads, only a few of the threads ever need to be active (i.e., require kernel resources, other than virtual memory) at the same instant.

Sometimes a particular thread must be visible to the system. For example, when a thread must support real-time response (e.g., mouse tracking thread) and thus be scheduled with respect to all the other execution entities in the system (i.e., global scheduling scope). The library accommodates this by allowing a thread to be permanently bound to an LWP. Even when a thread is bound to an LWP, it is still a thread and can interact or synchronize with other threads in the process, bound or unbound.

By defining both levels of interface in the architecture, we make clear the distinction between what the programmer sees and what the kernel provides. Most programmers program using threads and do not think about LWPs. When it is appropriate to optimize the behavior of the program, the programmer has the ability to tune the relationship between threads and LWPs. This allows programmers to structure their application assuming extremely lightweight threads while bringing the appropriate degree of kernel-supported concurrency to bear on the computation. A threads programmer can think of LWPs used by the application as the degree of real concurrency that the application requires.

LWP interfaces

LWPs are like threads. They share most of the process resources. Each LWP has a private set of registers and a signal mask. LWPs also have

[1]Allowing each thread to have its own vector of signal handlers was rejected because it would add a non-trivial amount of storage to each thread and it allowed the possibility of conflicting requests by different threads (e.g., `SIG_DFL` and a handler) that would make both implementation and use difficult. In addition, per-thread can be programmed on top of a shared vector of handlers, if required.

[2]The LWPs in this document are fundamentally different than the LWP library in SunOS 4.0. Lack of imagination and a desire to conform to generally accepted terminology lead us to use the same name.

[3]Actually, the SunOS 5.0 kernel schedules kernel threads on top of processors. Kernel threads are used for the execution context underlying LWPs. See [Eykholt 1992].

attributes that are unavailable to threads. For example they have a kernel-supported scheduling class, virtual time timers, an alternate signal stack and a profiling buffer.

The system interface to LWPs[4] is shown in Figure 2.

```
int      _lwp_create(context, flag, lwpidp)
void     _lwp_makecontext(ucp, func, arg, private,
                          stack, size);
void    *_lwp_getprivate();
void     _lwp_setprivate(ptr);
void     _lwp_exit();
int      _lwp_wait(waitfor, departedp);
lwpid_t _lwp_self();
int      _lwp_suspend(lwpid);
int      _lwp_continue(lwpid);
void     _lwp_mutex_lock(lwp_mutexp);
void     _lwp_mutex_unlock(lwp_mutex);
int      _lwp_cond_wait(lwp_condp, lwp_mutexp);
int      _lwp_cond_timedwait(lwp_condp, lwp_mutexp,
                             timeout);
int      _lwp_cond_signal(lwp_condp);
int      _lwp_cond_broadcast(lwp_condp);
int      _lwp_sema_wait(lwp_semap);
int      _lwp_sema_post(lwp_semap);
int      _lwp_kill(lwpid, sig);
```

Figure 2: LWP Interfaces

The `_lwp_create()` interface creates another LWP within the process. The `_lwp_makecontext()` interface creates a machine context that emulates the standard calling sequence to a function. The machine context can then be passed to `_lwp_create()`. This allows some degree of machine independence when using the LWP interfaces.

The LWP synchronization interfaces implement mutual exclusion locks, condition variables and counting semaphores. Variants are allowed that can support priority inversion prevention protocols such as priority inheritance. Counting semaphores are also provided to ensure that a synchronization mechanism that is safe with respect to asynchronous signals is available. In general, these routines only actually enter the kernel when they have to. For example, in some cases acquiring an LWP mutex does not require kernel entry if there is no contention. The kernel has no knowledge of LWP synchronization variables except during actual use.

The LWP synchronization variables are placed in memory by the application. If they are placed in shared memory or in mapped files that are accessible to other processes, they will synchronize LWPs between all the mapping processes even though the LWPs in different processes are generally invisible to each other. An advantage of using memory mapped files is that the synchronization variables along with other shared data can be preserved in a file. The application can be restarted and continue using its shared synchronization variables without any initialization. When a LWP synchronization primitive causes the calling LWP to block, the LWP is then suspended on a kernel-supported sleep queue associated with the offset in the mapped file or the shared memory segment[5] [6]. This allows LWPs in different processes to synchronize even though they have the synchronization variable mapped at different virtual addresses.

[4]The LWP interfaces described here are compatible with the current UNIX International LWP interfaces.

Both the `_lwp_getprivate()` and `_lwp_setprivate()` interfaces provide one pointer's worth of storage that is private to the LWP. Typically, a threads package can use this to point to its thread data structure. On SPARC this interface sets and gets the contents of register %g7[7].

An alternative to this approach is to have a private memory page per LWP. This requires more kernel effort and perhaps more memory requirements for each LWP. However, it is probably the most reasonable choice on register constrained machines, or on machines where user registers have not been reserved to the system.

The kernel also provides two additional signals. The first, SIGLWP is simply a new signal reserved for threads packages. It is used as an inter-LWP signal mechanism when directed to particular LWPs within the process via the `_lwp_kill()` interface. The second signal is SIGWAITING. This is currently generated by the kernel when it detects that all the LWPs in the process have blocked in indefinite waits. It is used by threads packages to ensure that processes don't deadlock indefinitely due to lack of execution resources.

Threads Library Implementation

Each thread is represented by a thread structure that contains the thread ID, an area to save the thread execution context, the thread signal mask, the thread priority, and a pointer to the thread stack. The storage for the stack is either automatically allocated by the library or it is passed in by the application on thread creation. Library allocated stacks are obtained by mapping in pages of anonymous memory. The library ensures that the page following the stack is invalid. This represents a "red zone" so that the process will be signalled if a thread should run off the stack. If the application passed in its own stack storage, it can provide a red zone or pack the stacks in its own way.

[5]Multics parlance used to call this the virtual address and what most people today call the virtual address (the effective addresses computed by the instructions) was called the logical address,

[6]Internally, the kernel uses a hash queue to synchronize LWPs. The hash lookup is based on a two word value that represents the internal file handle (vnode) and offset.

[7]On SPARC, register %g7, is reserved to the system in the Application Binary Interface [USO 1990]. ABI compliant applications cannot use it.

When a thread is created, a thread ID is assigned. In an earlier threads library implementation, the thread ID was a pointer to the thread structure. This implementation was discarded in favor of one where the thread ID was used as an index in a table of pointers to thread structures. This allows the library to give meaningful errors when a thread ID of an exited and deallocated thread is passed to the library.

Thread-local Storage

Threads have some private storage (in addition to the stack) called thread-local storage (TLS). Most variables in the program are shared among all the threads executing it, but each thread has its own copy of thread-local variables. Conceptually, thread-local storage is unshared, statically allocated data. It is used for thread-private data that must be accessed quickly. The thread structure itself and a version of `errno` that is private to the thread is allocated in TLS.

Thread-local storage is obtained via a new `#pragma, unshared`, supported by the compiler and linker. The contents of TLS are zeroed, initially; static initialization is not allowed. The size of TLS is computed by the run-time linker at program start time by summing the sizes of the thread-local storage segments of the linked libraries and computing offsets of the items in TLS[8]. The linker defines a symbol called `_etls` that represent the size of TLS. The threads library uses this at thread creation time to allocate space for TLS from the base of the new thread's stack. After program startup, the size of TLS is fixed and can no longer grow. This restricts programmatic dynamic linking (i.e., `dlopen()`) to libraries that do not contain TLS. Because of these restrictions, thread-local storage is not an exported interface. Instead a programmatic interface called thread-specific data [POSIX 1992] is available.

On SPARC, global register %g7 is assumed by the compiler to point to the base address of TLS. This is the same register used by the LWP private storage interfaces. The compiler generates code for TLS references relative to %g7. The threads library ensures that %g7 is set to the base address of TLS for the currently executing thread. The impact of this on thread scheduling is minimal.

Thread Scheduling

The threads library implements a thread scheduler that multiplexes thread execution across a pool of LWPs. The LWPs in the pool are set up to be nearly identical. This allows any thread to execute on any of the LWPs in this pool. When a thread executes, it is attached to an LWP and has all the attributes of being a kernel-supported thread.

All runnable, unbound, threads are on a user level, prioritized dispatch queue. Thread priorities range from 0 to "infinity"[9] A thread's priority is fixed in the sense that the threads library does not change it dynamically as in a time shared scheduling discipline. It can be changed only by the thread itself or by another thread in the same process. The unbound thread's priority is used only by the user level thread scheduler and is not known to the kernel.

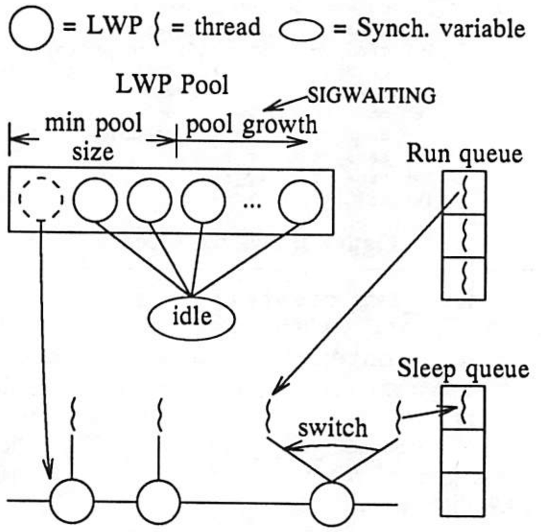


Figure 3: Thread Scheduling

Figure 3 gives an overview of the multiplexed scheduling mechanism. An LWP in the pool is either idling or running a thread. When an LWP is idle it waits on a synchronization variable (actually it can use one of two, see below) for work to do. When a thread, T1 is made runnable, it is added to the dispatch queue, and an idle LWP L1 (if one exists) in the pool is awakened by signalling the idle synchronization variable. LWP L1 wakes up and switches to the highest priority thread on the dispatch queue. If T1 blocks on a local synchronization object (i.e., one that is not shared between processes), L1 puts T1 on a sleep queue and then switches to the highest priority thread on the dispatch queue. If the dispatch queue is empty, the LWP goes back to idling. If all LWPs in the pool are busy when T1 becomes runnable, T1 simply stays on the dispatch queue, waiting for an LWP to become available. An LWP becomes available either when a new one is added to the pool or when one of the running threads blocks on a process-local synchronization variable, exits or stops, freeing its LWP.

[8]The size of TLS and offsets in TLS are computed at startup time rather than link time so that the amount of thread-local storage required by any particular library is not compiled into the application binary and the library may change it without breaking the binary interface.

[9]Currently, "infinity" is the maximum number representable by 32 bits.

Thread States and the Two Level Model

An unbound thread can be in one of five different states: RUNNABLE, ACTIVE, SLEEPING, STOPPED, and ZOMBIE. The transitions between these states and the events that cause these transitions are shown in Figure 4.

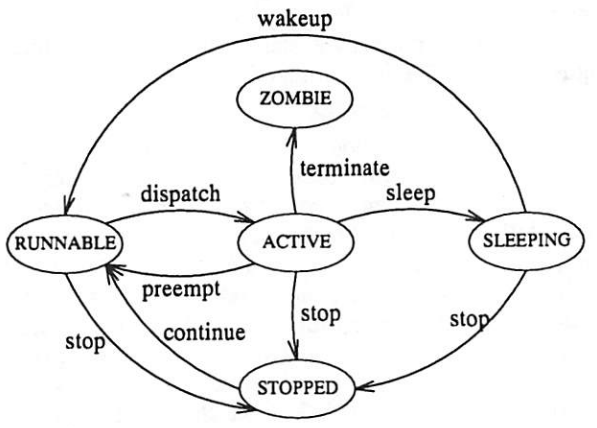


Figure 4: Thread states

While a thread is in the ACTIVE state, its underlying LWP can be running on a processor, sleeping in the kernel, stopped, or waiting for a processor on the kernel's dispatch queue. The four LWP states and the transitions between them can be looked upon as a detailed description of the thread ACTIVE state. This relationship between LWP states and thread states is shown in Figure 5.

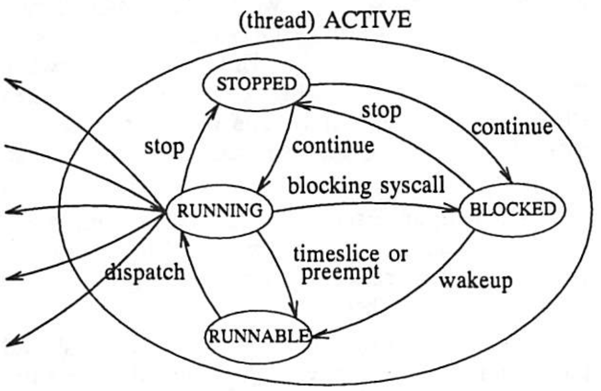


Figure 5: LWP states

Idling and parking

When an unbound thread exits and there are no more RUNNABLE threads, the LWP that was running the thread switches to a small idle stack associated with each LWP and *idles* by waiting on a global LWP condition variable. When another thread becomes RUNNABLE the global condition variable is signaled, and an idling LWP wakes up and attempts to run any RUNNABLE threads.

When a bound thread blocks on a process-local synchronization variable, its LWP must also stop running. It does so by waiting on a LWP semaphore associated with the thread. The LWP is now *parked.* When the bound thread unblocks, the parking semaphore is signalled so that the LWP can continue executing the thread.

When an unbound thread becomes blocked and there are no more RUNNABLE threads, the LWP that was running the thread also *parks* itself on the thread's semaphore, rather than *idling* on the idle stack and global condition variable. This optimizes the case where the blocked thread becomes runnable quickly, since it avoids the context switch to the idle stack and back to the thread.

Preemption

Threads compete for LWPs based on their priorities just as kernel threads compete for CPUs. A queue of ACTIVE threads is maintained. If there is a possibility that a RUNNABLE thread has a higher priority than that of some ACTIVE thread, the ACTIVE queue is searched to find such a thread. If such a thread is found, then this thread is removed from the queue and preempted from its LWP. This LWP then schedules another thread which is typically the higher priority RUNNABLE thread which caused the preemption.

There are basically two cases when the need to preempt arises. One is when a newly RUNNABLE thread has a higher priority than that of the lowest priority ACTIVE thread and the other is when the priority of an ACTIVE thread is lowered below that of the highest priority RUNNABLE thread.

ACTIVE threads are preempted by setting a flag in the thread's thread structure and then sending its LWP a SIGLWP. The threads library always installs its own handler for SIGLWP. When an LWP receives the signal, the handler checks if the current thread has a preemption posted. If it is, it clears the flag and then switches to another thread.

One side-effect of preemption is that if the target thread is blocked in the kernel executing a system call, it will be interrupted by SIGLWP, and the system call will be re-started when the thread resumes execution after the preemption. This is only a problem if the system call should not be re-started.

The Size of the LWP Pool

By default, the threads library automatically adjusts the number of LWPs in the pool of LWPs that are used to run unbound threads. There are two main requirements in setting the size of this pool: First, it must not allow the program to deadlock due to lack of LWPs. For example if there are more runnable unbound threads than LWPs and all the active threads block in the kernel in indefinite waits (e.g., read a tty), then the process can make no further progress until one of the waiting threads returns. The second requirement is that the library should make efficient use of LWPs. If the library were to create one LWP for every thread, in many cases these LWPs would simply be idle and the operating system would be overloaded by the resource requirements of many more LWPs than are actually being used. The advantages of a two-level model would be lost.

Let *C* be the total number of ACTIVE and RUNNABLE threads in an application, at any instant. In many cases, the value of *C* will fluctuate, sometimes quite dramatically as events come in and are processed, or threads synchronize. Ideally, if the time for creating or destroying an LWP were zero, the optimal design would be for the threads library to keep the size of the LWP pool equal to *C* at all times. This design would meet both requirements for the pool perfectly. In reality, since adjusting the size of the LWP pool has a cost, the threads library does not attempt to match it perfectly with *C* since this would be inefficient.

The current threads library implementation starts by guaranteeing that the application does not deadlock. It does so by using the new SIGWAITING signal that is sent by the kernel when all the LWPs in the process block in indefinite waits. The threads library ignores SIGWAITING by default. When the number of threads exceeds the number of LWPs in the pool, the threads library installs a handler for SIGWAITING. If the threads library receives a SIGWAITING and there are runnable threads, it creates a new LWP and adds it to the pool. If there are no runnable threads at the time the signal is received and the number of threads is less than the number of LWPs in the pool, it disables SIGWAITING.

A more aggressive implementation might attempt to compute a weighted time average of the application's concurrency requirements, and adjust the pool of LWPs more aggressively. Currently, it is not known whether the advantages of doing this would outweigh the extra overhead of creating more LWPs.

The application sometimes knows its concurrency requirements better than the threads library. The threads library provides an interface, `thr_setconcurrency()` that gives the library a hint as to what the expected concurrency level is. For example, a multi-threaded file copy program, with one input and one output thread might set its concurrency level to two; a multi-threaded window system server might set its concurrency level to the number of expected simultaneously active clients times the number of threads per client. If *n* is the level of concurrency given in `thr_setconcurrency()`, the threads library will ensure there are at least *n* LWPs in the pool when the number of threads is greater than or equal to *n*. If the number of threads is less than *n*, the library ensures that the number of LWPs is at least equal to the number of threads. Any growth in the number of LWPs past *n* is due to receiving a SIGWAITING.

The number of LWPs in the pool can grow to be greater than the number of threads currently in the process due to previous receptions of SIGWAITING or uses of `thr_setconcurrency()`. This can result in an excessive number of LWPs in the pool. The library therefore "ages" LWPs; they are terminated if they are unsued for a "long" time, currently five minutes. This is implemented by setting a per-LWP timer whenever the LWP starts idling. If the timer goes off, the LWP is terminated.

Mixed Scope Scheduling

A mixture of bound and unbound threads can coexist in the same process. Applications such as window systems can benefit from such mixed scope scheduling. A bound thread in the real-time scheduling class can be devoted to fielding mouse events which then take precedence over all other unbound threads which are multiplexing over time-sharing LWPs. When a bound thread calls the system's priority control interface, `priocntl()`, it affects the LWP it is bound to, and thus the thread itself. Thus, bound, real-time threads can coexist with unbound threads multiplexing across time-shared LWPs. Unbound threads continue to be scheduled in a multiplexed fashion in the presence of bound threads.

Reaping bound/unbound threads

When a detached thread (bound or unbound) exits, it is put on a single queue, called `deathrow` and their state is set to ZOMBIE. The action of freeing a thread's stack is not done at thread exit time to minimize the cost of thread exit by deferring unnecessary and expensive work. The threads library has a special thread called the reaper whose job is to do this work periodically. The reaper runs when there are idle LWPs, or when a reap limit is reached (the `deathrow` gets full). The reaper traverses the `deathrow` list, freeing the stack for each thread that had been using a library allocated stack.

When an undetached thread (bound or unbound) exits, it is added to the zombie list. Threads on the zombie list are reaped by the thread that executes `thr_join()` on them.

Since the reaper runs at high priority, it should not be run too frequently. Yet, it should not be run too rarely, since the rate at which threads are reaped has an impact on the speed of thread creation. This is because the reaper puts the freed stacks on a cache of available thread stacks, which speeds up stack allocation for new threads, an otherwise expensive operation.

Thread Synchronization

The threads library implements two basic types of synchronization variables, process-local (the default) or process-shared. In both cases the library ensures that the synchronization primitives themselves are safe with respect to asynchronous signals and therefore they can be called from signal handlers.

Process-local Synchronization Variables

The default blocking behavior is to put the thread to sleep. Each synchronization variable has a sleep queue associated with it. The synchronization primitives put the blocking threads on the synchronization variable's sleep queue and surrenders the executing thread to the scheduler. If the thread is unbound, the scheduler dispatches another thread to the thread's underlying LWP. A Bound thread, however, must stay permanently bound to its LWP so the LWP is *parked* on its thread.

Blocked threads are awakened when the synchronization variables become available. The synchronization primitives that release blocked threads first check if threads are waiting for the synchronization variables. A blocked thread is removed from the synchronization variable's sleep queue and is dispatched by the scheduler. If the thread is bound, the scheduler unparks the bound thread so that its LWP is dispatched by the kernel. For unbound threads, the scheduler simply places the thread on a run queue corresponding to its priority. The thread is then dispatched to an LWP in priority order.

Process-shared Synchronization variables

Process-shared synchronization objects can also be placed in memory that is accessible to more than one process and can be used to synchronize threads in different processes. Process-shared synchronization variables must be initialized when they are created because their blocking behavior is different from the default. Each synchronization primitive supports an initialization function that must be called to mark the process-shared synchronization variable as process-shared. The primitives can then recognize the synchronization variables that are shared and provide the correct blocking behavior. The primitives rely on LWP synchronization primitives to put the blocking threads to sleep in the kernel still attached to their LWPs, and to correctly synchronize between processes.

Signals

The challenge in providing the SunOS MT signal semantics for user threads in a two-level model of multi-threading is that signals are sent by the kernel but user level threads and their masks are invisible to the kernel. In particular, since signal delivery to a thread is dependent on the thread signal mask, the challenge is to elicit the correct program behavior even though the kernel cannot make the correct signalling decisions because it cannot see all the masks.

The implementation has the additional goal of providing cheap async safe synchronization primitives. A function is said to be async safe if it is reentrant with respect to signals, i.e., it is callable from a signal handler invoked asynchronously. Low overhead async safe synchronization primitives are crucial for multi-threaded libraries containing internal critical sections, such as the threads library and its clients, such as libc, etc. For example, consider a call to the threads library, say `mutex_lock()`, which is interrupted asynchronously while holding an internal threads library lock, *L*. The asynchronous handler could potentially call into `mutex_lock()` also and try to acquire *L* again, resulting in a deadlock. One way of making `mutex_lock()` async safe is to mask signals while in *L*'s critical section. Thus, efficient signal masking was an important goal since it could provide efficient async safe critical sections.

Signal Model Implementation

One implementation strategy would be for each LWP that is running a thread to reflect the thread's signal mask. This allows the kernel to directly choose a thread to signal from among the ACTIVE threads within a process. The signals that a process can receive changes as the threads in the application cycle through the ACTIVE state.

This strategy has a problem with threads that are rarely ACTIVE and are the only threads in the application that have certain signals enabled. These threads are essentially asleep waiting to be interrupted by a signal which they will never receive. In addition, a system call must be done whenever an LWP switches between threads having different masks or when an active thread adjusts its mask.

This problem can be solved if the LWP signal masks and the ACTIVE thread signal masks are treated more independently. The set of signals that a process can receive is equal to the intersection of all the thread signal masks. The library ensures that the LWP signal mask is either equal to the thread mask or it is less restrictive. This means occasionally signals are sent by the kernel to ACTIVE threads that have the signal disabled.

When this occurs the threads library prevents the signal from actually reaching the interrupted thread by interposing its own signal handlers below the application signal handlers. When a signal is delivered, the global handler checks the current thread's signal mask to determine if the thread can receive this signal. If the signal is masked, the global handler sets the current LWP's signal mask to the current thread's signal mask. Then the signal is resent to the process if it is an undirected signal or to its LWP if it is a directed signal. The kernel signal delivery mechanism provides information that allows the signal handler to distinguish between directed and undirected signals. If the signal is not masked, the global handler calls the signal's application handler. If the signal is not appropriate for any of the currently `ACTIVE` threads, the global handler can cause one of the inactive threads to run if it has the signal unmasked.

Synchronously generated signals are simply delivered by the kernel to the ACTIVE thread that caused them.

Sending a directed signal

A thread can send a signal to another thread in the same process using `thr_kill()`. The basic means of sending a signal to a thread is to send it to the LWP it runs on. If the target thread is not ACTIVE, `thr_kill()` just posts the signal on the thread in a pending signals mask. When the target thread resumes execution, it receives the pending signals[10]. If the target thread is ACTIVE, `thr_kill()` sends the signal to the target thread's LWP via `lwp_kill()`. If the target LWP is blocking the signal (which implies that the thread is blocking it too), the signal stays pending on the LWP, until the thread unblocks it. While the signal is pending on the LWP, the thread is temporarily bound to the LWP until the signals are delivered to the thread.

Signal Safe Critical Sections

As stated previously, to prevent deadlock in the presence of signals, critical sections that are reentered in a signal handler in both multi-threaded applications and the threads library should be safe with respect to signals. All asynchronous signals should be masked during such critical sections.

The threads library signals implementation allows multi-threaded applications to make critical sections safe as efficiently as possible via a low overhead implementation of the `thr_sigsetmask()` interface. If signals do not occur, `thr_sigsetmask()` does not result in a system call. In this case, it is as fast as just modifying the user-level thread signal mask.

The threads library has an even faster means of achieving signal safety for its internal critical sections. The threads library sets/clears a special flag in the threads structure whenever it enter/exits an internal critical section. Effectively, this flag serves as a signal mask to mask out all signals. If the flag is set when a signal is delivered, the threads library global signal handler will defer the signal as described in the above section.

Debugging Threads

A debugger that can control library supported threads requires access to information about the threads inside the debugged process. The normal kernel-supported debugging interfaces (/proc) are insufficient. One could build complete knowledge of the threads implementation into the debugger, but that would force a re-release of the debugger whenever some internal threads library data structure changes. Instead the threads library provides a separate dynamically linked thread debugging library that the debugger links with via `dlopen()`. The thread debugging library contains interfaces that allow the debugger access to general thread information, without the debugger itself containing knowledge of the threads library implementation. All the threads library specific knowledge is contained in the thread debugging library. The interfaces in the thread debugging library allow the debugger to for example, enumerate the list of threads in the debugged process, or get/set the state and/or registers of each thread (ACTIVE or not).

We have a version of DBX with thread extensions that dynamically links with this library. The new features allow the programmer to debug a multi-thread process. It will list all threads in the process and their states, whether they are sleeping, running, active, stopped or zombied. DBX can change its focus to any thread so that its internal state can be analyzed with the standard DBX commands.

References

[Cooper 1990] E. C. Cooper, R. P. Draves, "C Threads", Department of Computer Science, Carnegie Mellon University, September 1990.

[Eykholt 1992] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams, "Beyond Multiprocessing: Multithreading the SunOS Kernel", Proc. 1992 USENIX Summer Conference, pp. 11-18.

[Faulkner 1991] R. Faulkner, R. Gomes, "The Process File System and Process Model in UNIX System V", Proc. 1991 USENIX Winter Conference.

[Golub 1990] D. Golub, R. Dean, A. Florin, R. Rashid, "UNIX as an Application Program", Proc. 1990 USENIX Summer Conference, pp 87-95.

[Khanna 1992] Sandeep Khanna, Michael Sebrée, John Zolnowsky, "Realtime Scheduling in SunOS 5.0", Proc. 1992 USENIX Winter Conference.

[POSIX 1992] POSIX P1003.4a, "Threads Extension for Portable Operating Systems", IEEE.

[Powell 1991] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, "SunOS Multi-thread Architecture", Proc. 1991 USENIX Winter Conference.

[Sha 1990] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Realtime Synchronization", Vol 39, No 9, September 1990, IEEE Transactions on Computers.

[10]The threads library context switch code ensures this by sending the pending signals to the LWP that resumes the thread.

[USO 1990] UNIX Software Operation, ''System V Application Binary Interface, SPARC Processor Supplement'', UNIX Press.

Author Information

Dan Stein is currently a member of technical staff at SunSoft, Inc. where he is one of the developers of the SunOS mult-thread Architecture. He graduated from the University of Wisconsin in 1981 with a B.S. in Computer Science.

Devang Shah is currently a Member of Technical Staff at SunSoft, Inc. He received an M.A. in Computer Science from the University of Texas at Austin in 1989 and a B. Tech. in Electronics Engg. from the Institute of Technology, B.H.U., India in 1985. At UT-Austin he extended SunOS 3.2 to provide lightweight processes. Prior to UT-Austin, he worked at Tata Consultancy Services, Bombay.

Beyond Multiprocessing: Multithreading the SunOS Kernel

J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams – SunSoft, Inc.

ABSTRACT

Preparing the SunOS/SVR4 kernel for today's challenges: symmetric multiprocessing, multi-threaded applications, real-time, and multimedia, led to the incorporation of several innovative techniques. In particular, the kernel was re-structured around threads. Threads are used for most asynchronous processing, including interrupts. The resulting kernel is fully preemptible and capable of real-time response. The combination provides a robust base for highly concurrent, responsive operation.

Introduction

When we started to investigate enhancements to the SunOS kernel to support multiprocessors, we realized that we wanted to go further than merely adding locks to the kernel and keeping the user process model unchanged. It was important for the kernel to be capable of a high degree of concurrency on tightly coupled symmetric multiprocessors, but it was also a goal to support more than one thread of control within a user process. These threads must be capable of executing system calls and handling page faults independently. On multiprocessor systems, these threads of control must be capable of running concurrently on different processors. [Powell 1991] described the user-visible thread architecture.

We also wanted the kernel to be capable of bounded dispatch latency for real-time threads [Khanna 1992]. Real-time response requires absolute control over scheduling, requiring preemption at almost any point in the kernel, and elimination of unbounded priority inversions wherever possible.

The kernel itself is a very complex multithreaded program. Threads can be used by user applications as a structuring technique to manage multiple asynchronous activities; the kernel benefits from a thread facility that is essentially the same.

The resulting SunOS 5.0 kernel, the central operating system component of Solaris 2.0, is fully preemptible, has real-time scheduling, symmetrically supports multiprocessors, and supports user-level multithreading. Several of the locking strategies used in this kernel were described in [Kleiman 1992]. In this paper we'll describe some of the implementation features that make this kernel unique.

Overview of the Kernel Architecture

A kernel thread is the fundamental entity that is scheduled and dispatched onto one of the CPUs of the system. A kernel thread is very lightweight, having only a small data structure and a stack. Switching between kernel threads does not require a change of virtual memory address space information, so it is relatively inexpensive. Kernel threads are fully preemptible and may be scheduled by any of the scheduling classes in the system, including the real-time (fixed priority) class. Since all other execution entities are built using kernel threads, they represent a fully preemptible, real-time "nucleus" within the kernel.

Kernel threads use synchronization primitives that support protocols for preventing priority inversion, so a thread's priority is determined by which activities it is impeding by holding locks as well as by the service it is performing [Khanna 1992].

SunOS uses kernel threads to provide asynchronous kernel activity, such as asynchronous writes to disk, servicing STREAMS queues, and callouts. This removes various diversions in the idle loop and trap code and replaces them with independently scheduled threads. Not only does this increase potential concurrency (these activities can be handled by other CPUs), but it also gives each asynchronous activity a priority so that it can be appropriately scheduled.

Even interrupts are handled by kernel threads. The kernel synchronizes with interrupt handlers via normal thread synchronization primitives. If an interrupt thread encounters a locked synchronization variable, it blocks and allows the critical section to clear.

A major feature of the new kernel is its support of multiple kernel-supported threads of control, called lightweight processes (LWPs), in any user process, sharing the address space of the process and other resources, such as open files. The kernel supports the execution of user LWPs by associating a kernel thread with each LWP, as shown in Figure 1. While all LWPs have a kernel thread, not all kernel threads have an LWP.

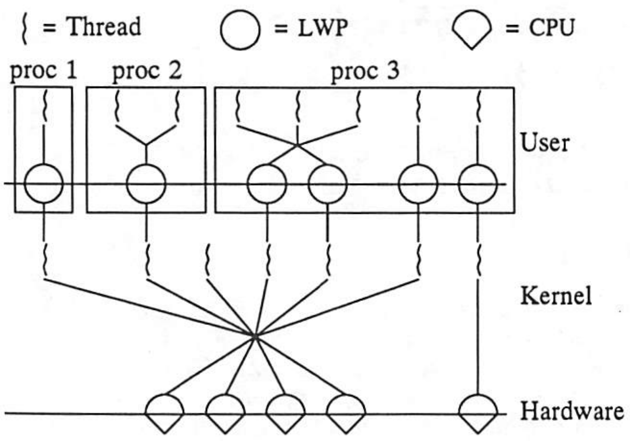


Figure 1: Multi-thread architecture examples

A user-level library uses LWPs to implement user-level threads [Stein 1992]. These threads are scheduled at user-level and switched by the library to any of the LWPs belonging to the process. User threads can also be bound to a particular LWP. Separating user-level threads from the LWP allows the user thread library to quickly switch between user threads without entering the kernel. In addition, it allows a user process to have thousands of threads, without overwhelming kernel resources.

Data Structures

In the traditional kernel, the `user` and `proc` structures contained all kernel data for the process. Processor data was held in global variables and data structures. The per-process data was divided between non-swappable data in the `proc` structure, and swappable data in the `user` structure. The kernel stack of the process, which is also swappable, was allocated with the `user` structure in the user area, usually one or two pages long.

The restructured kernel must separate this data into data associated with each LWP and its kernel thread, the data associated with each process, and the data associated with each processor. Figure 2 shows the relationship of these data structures in the restructured kernel.

The per-process data is contained in the `proc` structure. It contains a list of kernel threads associated with the process, a pointer to the process address space, user credentials, and the list of signal handlers. The `proc` structure also contains the vestigial `user` structure, which is now much smaller than a page, and is no longer practical to swap.

The LWP structure contains the per-LWP data such as the process-control-block (pcb) for storing user-level processor registers, system call arguments, signal handling masks, resource usage information, and profiling pointers. It also contains pointers to the associated kernel thread and process structures. The kernel stack of the thread is allocated with the LWP structure inside a swappable area.

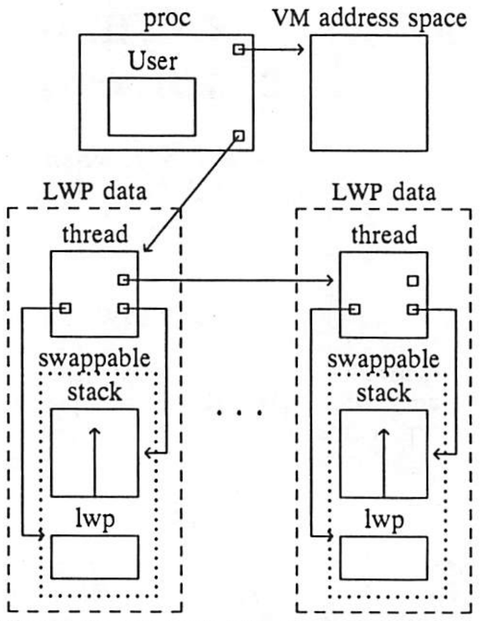


Figure 2: MT Data Structures for a Process

The kernel thread structure contains the kernel registers, scheduling class, dispatch queue links, and pointers to the stack and the associated LWP, process, and CPU structures. The thread structure is not swapped, so it also contains some data associated with the LWP that is needed even when the LWP structure is swapped out. Thread structures are linked on a list of threads for the process, and also on a list of all existing threads in the system.

Per-processor data is kept in the `cpu` structure, which has pointers to the currently executing thread, the idle thread for that CPU, and current dispatching and interrupt handling information. There is a substructure of the `cpu` structure that can be architecture dependent, but the main body is intended to be applicable to most multiprocessing architectures.

To speed access to the thread, LWP, process, and CPU structures, the SPARC implementation uses a global register, %g7, to point to the current thread structure. A C-preprocessor macro, `curthread`, allows access to fields in the current thread structure with a single instruction. The current LWP, process, and CPU structures are quickly accessible through pointers in the thread structure. In the future we may dedicate additional global registers for other frequently accessed structures.

Kernel Thread Scheduling

SunOS 5.0 provides several scheduling classes. A scheduling class determines the relative priority of processes within the class, and converts that priority to a global priority. With the addition of multithreading, the scheduling classes and dispatcher operate on threads instead of processes. The scheduling classes currently supported are system, timesharing, and real-time (fixed-priority).

The dispatcher chooses the thread with the greatest global priority to run on the CPU. If more than one thread has the same priority, they are dispatched in round-robin order.

The kernel has been made preemptible to better support the real-time class and interrupt threads. Preemption is disabled only in a small number of bounded sections of code. This means that a runnable thread runs as soon as is practical after its priority becomes high enough. For example, when thread A releases a lock on which higher priority thread B is sleeping, the running thread A immediately puts itself back on the run queue and allows the CPU to run thread B. On a multiprocessor, if thread A has better priority than thread B, but thread B has better priority than the current thread on another CPU, that CPU is directed to preempt its current thread and choose the best thread to run. In addition, user code run by an underlying kernel thread of sufficient priority (e.g., real-time threads) will execute even though other lower priority kernel threads wait for execution resources. Further details can be found in [Khanna 1992].

System Threads

System threads can be created for short or long-term activities. They are scheduled like any other thread, but usually belong to the system scheduling class. These threads have no need for LWP structures, so the thread structure and stack for these threads can be allocated together in a non-swappable area, as shown in Figure 3.

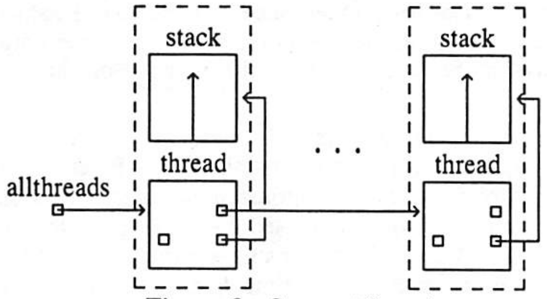


Figure 3: System Threads

A new segment driver, `seg_kp`, handles stack allocations. It handles virtual memory allocations for the kernel that can be paged or swapped out; it also provides "red zones" to protect against stack overflow. System threads use `seg_kp` for the stack and the thread structure, in a non-swappable region. LWPs use it to allocate the LWP structure and kernel stack in a swappable region.

Synchronization Architecture

The kernel implements the same synchronization objects for internal use as are provided by the user-level libraries for use in multithreaded application programs [Powell 1991]. These are mutual exclusion locks (*mutexes*), condition variables, semaphores, and multiple readers, single writer (readers/writer) locks. The interfaces are shown in Figure 4[1].

```
/* Mutual exclusion locks */
void    mutex_enter(kmutex_t *lp);
void    mutex_exit(kmutex_t *lp);

void    mutex_init(kmutex_t *lp, char *name,
            kmutex_type_t type, void *arg);
void    mutex_destroy(kmutex_t *lp);
int     mutex_tryenter(kmutex_t *lp);

/* condition variables */
void    cv_wait(kcondvar_t *cp, kmutex_t *lp);
int     cv_wait_sig(kcondvar_t *cp,
            kmutex_t *lp);
int     cv_timedwait(kcondvar_t *cvp,
            kmutex_t *lp, long tim);
void    cv_signal(kcondvar_t *cp);
void    cv_broadcast(kcondvar_t *cp);

/* multiple reader, single writer locks */
void    rw_init(krwlock_t *lp, char *name,
            krw_type_t type, void *arg);
void    rw_destroy(krwlock_t *lp);
void    rw_enter(krwlock_t *lp, krw_t rw);
int     rw_tryenter(krwlock_t *lp, krw_t rw);
void    rw_exit(krwlock_t *lp);
void    rw_downgrade(krwlock_t *lp);
int     rw_tryupgrade(krwlock_t *lp);

/* counting semaphores */
void    sema_init(ksema_t *sp,
            unsigned int val, char *name,
            ksema_type_t type, void *arg);
void    sema_destroy(ksema_t *sp);
void    sema_p(ksema_t *sp);
int     sema_p_sig(ksema_t *sp);
int     sema_tryp(ksema_t *sp);
void    sema_v(ksema_t *sp);
```

Figure 4: Kernel Thread Synchronization Interfaces

These are all implemented such that the behavior of the synchronization object is specified when it is initialized. Synchronization operations, such as acquiring a mutex lock, take a pointer to the object as an argument and may behave somewhat differently depending on the type and optional type-specific argument specified when the object was initialized.

Most of the synchronization objects have types that enable collecting statistics such as blocking counts or times. A patchable kernel variable can also set the default types to enable statistics gathering. This allows the selection of statistics gathering on particular synchronization objects or on the kernel as a whole.

[1]Note that kernel synchronization primitives must use a different type name than user synchronization primitives so that the types are not confused in applications that read internal kernel data structures.

The semantics of most of the synchronization primitives cause the calling thread to be prevented from progressing past the primitive until some condition is satisfied. The way in which further progress is impeded (e.g., sleep, spin, or other) is a function of the initialization. By default, the kernel thread synchronization primitives that can logically block, can *potentially* sleep.

Some of the synchronization primitives are strictly bracketing (e.g., the thread that *locks* a mutex must be the thread that *unlocks* it) and a single owner can be determined (i.e., mutexes and writer locks). In these cases, the synchronization primitives support the priority inheritance protocol, as described in [Khanna 1992].

Some synchronization primitives are intended for situations where they may block for long or indeterminate periods. Variants of some of the primitives are provided (e.g., `cv_wait_sig()` and `sema_p_sig()`) that allow blocking to be interrupted by a reception of a signal. There is no non-local jump to the head of the system call, as there was in the traditional `sleep` routine. When a signal is pending, the primitive returns with a value indicating the blocking was interrupted by a signal and the caller must release any resources and return.

Mutual Exclusion Lock Implementation.

Mutual exclusion locks (mutexes) prevent more than one thread from proceeding when the lock is acquired. They prevent races on access to shared data and are by far the most heavily used primitive.

Mutexes are usually held for short intervals. For example, it would not be good to hold a critical system mutex while waiting for disk I/O to complete. Mutexes are not recursive; the owner of the lock cannot again call *mutex_enter()* for the same lock. If a thread holds a mutex, the same thread must be the one to release the mutex. These rules are enforced to promote good programming practice and to avoid deadlocks.

If *mutex_enter* cannot set the lock (because it is already set), the blocking action taken depends on the mutex type that was passed to *mutex_init*, and stored in the mutex. The default blocking policy for mutexes, called *adaptive* (type `MUTEX_DEFAULT`), spins while the owner of the lock (recorded when the lock is acquired) remains running on a processor. This is done by polling the owner's status in the spin wait loop[2]. If the owner ceases to run, the caller stops spinning and sleeps[3]. This gives fast response and low overhead for simple contention.

Spin mutexes are available as type `MUTEX_SPIN`, which takes as its type-specific argument the interrupt level to be disabled while the mutex is held. It is rarely used, as adaptive mutexes are more efficient, in general.

Device drivers are restricted to using type `MUTEX_DRIVER`, which takes a Sun-DDI-defined opaque value as an argument. This argument is basically an interrupt priority in the current implementation, and determines whether the blocking policy is adaptive or spin, based on whether the interrupt priority is above the "thread level" (see below).

A simple trick speeds up `mutex_enter()` for adaptive mutexes. Non-adaptive mutexes use a separate primitive lock field in the mutex data structure, with the lock field used by the adaptive type always in the locked state. This is so that `mutex_enter()` can always attempt to apply an adaptive lock first, and only if that fails, consider the possibility that the mutex might be another type.

Turnstiles vs Queues in Synchronization Objects

Each synchronization object requires a way of finding threads that are suspended waiting for that object. It is important to keep the storage cost of synchronization objects small, because many system structures contain synchronization objects, so the queue header is not directly in the object. Instead, two bytes in the synchronization object are used to find a *turnstile* structure containing the sleep queue header and priority inheritance information [Khanna 1992]. Turnstiles are preallocated such that there are always more turnstiles than the number of threads active.

One alternative method would be to select the sleep queue from an array using a hash function on the address of the synchronization object. This is essentially the approach used by `sleep()` in the traditional kernel. The turnstile approach is favored for more predictable real-time behavior, since they are never shared by other locks, as hashed sleep queues sometimes are.

Interrupts as Threads

Many implementations [Hamilton 1988] [Peacock 1992] have a variety of synchronization primitives that have similar semantics (e.g., mutual exclusion) yet explicitly sleep or spin for blocking. For mutexes, the spin primitives must hold interrupt priority high enough while the lock is held to prevent any interrupt handlers that may also use the synchronization object from interrupting while the object is locked, causing deadlock. The interrupt level must be raised before the lock is acquired and then lowered after the lock is released.

[2]In order to avoid locking while inspecting the owner's status during the spin, the state is determined indirectly. The algorithm spins while the current thread pointer of any CPU points to the owning thread, indicating it is running.

[3]On uniprocessors, this turns into always sleeping, since the owner cannot be running.

This has several drawbacks. First, the raising and lowering of interrupt priority can be an expensive operation, especially on architectures that require external interrupt controllers (remember that mutexes are heavily used). Secondly, in a modular kernel, such as SunOS, many subsystems are interdependent. In several cases (e.g., mapping in kernel memory or memory allocation) these requests can come from interrupt handlers and can involve many kernel subsystems. This in turn, means that the mutexes used in many kernel subsystems must protect themselves at a relatively high priority from the *possibility* that they may be required by an interrupt handler. This tends to keep interrupt priority high for relatively long periods and the cost of raising and lowering interrupt priority must be paid for every mutex acquisition and release. Lastly, interrupt handlers must live in a constrained environment that avoids any use of kernel functions that can potentially sleep, even for short periods.

To avoid these drawbacks, the SunOS 5.0 kernel treats most interrupts as asynchronously created and dispatched high-priority threads. This enables these interrupt handlers to sleep, if required, and to use the standard synchronization primitives.

On most architectures putting threads to sleep must be done in software. This must be protected from interrupts if interrupts are to sleep themselves or wakeup other threads. The restructured kernel uses a primitive spin lock protected by raised priority to implement this. This is one of a few bounded sections of code where interrupts are locked out.

Traditional UNIX kernel implementations [Leffler 1989] [Bach 1986] also protect the dispatcher by locking out interrupts, usually all interrupts. The restructured kernel has a modifiable level (the "thread level") above which interrupts are no longer handled as threads and are treated more like non-portable "firmware" (e.g., simulating DMA via programmed I/O). These interrupt handlers can only synchronize using the spin variants of mutex locks and software interrupts. If the "thread level" is set to the maximum priority, then all interrupts are locked out during dispatching. For implementations where the "firmware" cannot tolerate even the relatively small dispatcher lockout time, the "thread level" can be lowered. Typically this is lowered to the interrupt level at which the scheduling clock runs.

Implementing Interrupts as Threads

Previous versions of SunOS have treated interrupts in the traditional UNIX way. When an interrupt occurs the interrupted process is held captive (*pinned*) until the interrupt returns. Typically, interrupts are handled on the kernel stack of the interrupted process or on a separate interrupt stack. The interrupt handler must complete execution and get off the stack before anything else is allowed to run on that processor. In these systems the kernel synchronizes with interrupt handlers by blocking out interrupts while in critical sections.

In SunOS 5.0, interrupts behave like asynchronously created threads. Interrupts must be efficient, so a full thread creation for each interrupt is impractical. Instead, we preallocate interrupt threads, already partly initialized. When an interrupt occurs, we do the minimum amount of work to move onto the stack of an interrupt thread, and set it as the current thread. At this point, the interrupt thread and the interrupted thread are not completely separated. The interrupt thread is not yet a full-fledged thread (it cannot be descheduled) and the interrupted thread is *pinned* until the interrupt thread returns or blocks, and cannot proceed on another CPU. When the interrupt returns, we restore the state of the interrupted thread and return.

Interrupts may nest. An interrupt thread may itself be interrupted and be pinned by another interrupt thread.

If an interrupt thread blocks on a synchronization variable (e.g., mutex or condition variable), it saves state (*passivates*) to make it a full-fledged thread, capable of being run by any CPU, and then returns to the pinned thread. Thus most of the overhead of creating a full thread is only done when the interrupt must block, due to contention[4].

While an interrupt thread is in progress, the interrupt level it is handling, and all lower-priority interrupts, must be blocked. This is handled by the normal interrupt priority mechanism unless the thread blocks. If it blocks, these interrupts must remain disabled in case the interrupt handler is not reenterable at the point that it blocked or it is still doing high-priority processing (i.e., should not be interrupted by lower-priority work). While it is blocked, the interrupt thread is bound to the processor it started on as an implementation convenience and to guarantee that there will always be an interrupt thread available when an interrupt occurs (though this may change in the future). A flag is set in the `cpu` structure indicating that an interrupt at that level has blocked, and the minimum interrupt level is noted. Whenever the interrupt level changes, the CPU's base interrupt level is checked, and the actual interrupt priority level is never allowed to be below that.

There is also an interface which allows an interrupt thread to continue as a normal, high-priority thread. When `release_interrupt()` is called, it saves the state of the the pinned thread and clears the indication that the interrupt thread has blocked, allowing the CPU to lower the interrupt

[4]On SPARC this overhead involves flushing the entire register file. This is only done if the interrupt handler sleeps, not during interrupt handling without contention.

priority level.

An alternative approach to this is to use bounded first-level interrupt handlers to capture device state and then wake up an interrupt thread that is waiting to do the remainder of the servicing [Barnett 1992]. This approach has the disadvantages of requiring device drivers to be restructured and of always requiring a full context switch to the second level thread. The approach used in SunOS 5.0 allows full thread behavior without restructured drivers and with very little additional cost in the no-contention case.

Interrupt Thread Cost

The additional overhead in taking an interrupt is about 40 SPARC instructions. The savings in the mutex enter/exit path is about 12 instructions. However, mutex operations are much more frequent than interrupts, so there is a net gain in time cost, as long as interrupts don't block too frequently. The work to convert an interrupt into a "real" thread is performed only when there is lock contention.

There is a cost in terms of memory usage also. Currently an interrupt thread is preallocated for each potentially active interrupt level below the thread level for each CPU[5]. An additional interrupt thread is preallocated for the clock (one per system). Since each thread requires a stack and a data structure, perhaps 8K bytes or so, the memory cost can be high.

However, it is unlikely that all interrupt levels are active at any one time, so it is possible to have a smaller pool of interrupt threads on each CPU and block all subsequent interrupts below the thread level when the pool is empty, essentially limiting how many interrupts may be simultaneously active.

Clock Interrupt

The clock interrupt[6] is handled specially. There is only one clock interrupt thread in the system (not one per CPU), and the clock interrupt handler invokes the clock thread only if it is not already active.

The clock thread could possibly be delayed for more than one clock tick by blocking on a mutex or by higher-level interrupts. When a clock tick occurs and the clock thread is already active, the interrupt is cleared and a counter is incremented. If the clock thread finds the counter non-zero before it returns, it will decrement the counter and repeat the clock processing. This occurs very rarely in practice. When it occurs, it is usually due to heavy activity at higher interrupt levels. It can also occur while debugging.

[5]There are nine interrupt levels on the Sun SPARC implementation that can potentially use threads.

[6]This occurs 100 times a second on current Sun SPARC implementations.

Kernel Locking Strategy

The locking approach used almost exclusively in the kernel to ensure data consistency is data-based locking. That is, the mutex and readers/writer locks each protect a set of shared data, as opposed to protecting routines (monitors). Every piece of shared data is protected by a synchronization object.

Some aspects of locking in the virtual memory, file system, STREAMS, and device drivers have already been discussed in [Kleiman 1992]. Here we'll elaborate a bit on device driver issues, as they are closely related to interrupt threads.

Non-MT Driver Support

Some drivers haven't been modified to protect themselves against concurrency in a multithreaded environment. These drivers are called *MT-unsafe*, because they don't provide their own locking.

In order to provide some interim support for *MT-unsafe* drivers, we provided wrappers that acquire a single global mutex, `unsafe_driver`. These wrappers insure that only one such driver will be active at any one time. This wrapper is illustrated by Figure 5.

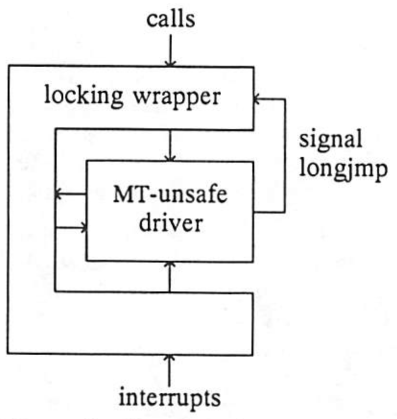


Figure 5: Unsafe Driver Wrapper

There are several ways a driver may be entered, from the explicit driver entry points, interrupts, and call-backs. Each of these entries must acquire the `unsafe_driver` mutex if the driver isn't safe. For example, if an MT-unsafe driver uses `timeout()` to request a function call at a later time, the `callout` structure is marked so that the `unsafe_driver` mutex will be held during the function call.

MT-unsafe drivers can also use the old `sleep/wakeup` mechanism. `Sleep()` safely releases the `unsafe_driver` mutex after the thread is asleep, and reacquires it before returning.

The `longjmp()` feature of `sleep()` is maintained as well. When a thread is signalled in `sleep()`, if it specified a dispatch value greater than PZERO, a `longjmp()` takes the thread to a `setjmp()` that was performed in the unsafe driver entry wrapper, which returns EINTR to the caller of the driver.

`Sleep()` checks to make sure it is called by an MT-unsafe driver, and panics if it isn't. It isn't safe to use `sleep()` from a driver which does its own locking.

It is fairly easy to provide at least simple locking for a driver, so almost all drivers in the system have some of their own locking. These drivers are called *MT-safe*, regardless of how fine-grained their locking is. Some developers have used the term *MT-hot* to indicate that a driver does fine-grained locking.

SVR4/MP DKI Locking Primitives

As we implemented our driver interfaces, UNIX International and USL were defining the SVR4 Multiprocessor Device Driver Interface and Driver-Kernel Interface (DDI/DKI), with a different set of locking primitives, based around the traditional UNIX interrupt-blocking model.

SunOS 5.0 implements those interfaces to the extent defined so far, using our locking primitives and ignoring any spin semantics. This allows drivers using those interfaces to be more easily ported. SunOS drivers typically use the the SunOS synchronization primitives.

Implementation Technology

Some interesting techniques made it easier to get this all working.

Kernel Time Slicing

Since the kernel is fully preemptible we were able to make kernel threads time-slice. We simply added code to the clock interrupt handler to preempt whatever thread was interrupted. This allows even a uniprocessor to have almost arbitrary code interleavings. Increasing the clock interrupt rate made this even more valuable in finding windows where data was unprotected. By causing kernel threads to preempt each other as often as possible we were able to find locking problems using uniprocessor hardware before multiprocessor hardware was available. Even when working multiprocessor hardware arrived, there were far more uniprocessors available than multiprocessors. We intend this only as a debugging feature, since it does have some adverse performance impact, however slight.

Lock Hierarchy Violation Detection

Instead of establishing a system lock hierarchy *a priori*, we developed a static analysis tool that would check for lock ordering violations in the system. This lint-like tool, called *locknest*, reads C source code, constructs call graphs and reports on locking cycles. We feel it helped during early implementation debugging, and probably reduced the amount of time spent debugging deadlocks. A similar tool is described in [Korty 1989].

Deadlock Detection

A side-benefit of the priority inheritance mechanism [Khanna 1992], is that deadlocks caused by hierarchy violations are usually detected at run time as well. It does a good job on mutexes and readers/writer locks held for write, but since there isn't a complete list of threads holding a read lock, it can't always find deadlocks involving readers/writer locks. There are other deadlocks possible with condition variables; these aren't detected.

Summary

SunOS 5.0 is a multithreaded and symmetric multiprocessor version of the SVR4 kernel. The primary features are:

- Fully preemptible, real-time kernel
- High degree of concurrency on symmetric multiprocessors
- Support for user threads
- Interrupts handled as independent threads
- Adaptive mutual-exclusion locks

The thread models inside the kernel and at user level are almost identical. The scheduling of kernel threads onto CPUs is analogous to the way the threads library schedules user-level threads onto LWPs. The use of threads for structuring the kernel has mostly good effects though they can be overused. Threads do have a cost. The stacks are large, and must be allocated on separate pages if protection for potential stack overrun is needed. Also, context switching is still expensive. Some things are still better implemented by callouts and other ''zero-weight'' processes, but threads provide a nice structuring paradigm for the kernel.

References

[Bach 1986] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[Barnett 1992] David Barnett, *Kernel Threads and their Performance Benefits*, Real Time, Vol. 4, No. 1, Lynx Real-Time Systems, Inc., Los Gatos, CA., 1992.

[Hamilton 1988] Graham Hamilton and Daniel S. Conde, *An Experimental Symmetric Multiprocessor Ultrix Kernel*, USENIX, Winter 1988, Dallas, Texas.

[Kleiman 1992] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner, *Symmetric Multiprocessing in Solaris 2.0*, COMPCON Spring 1992, p181, San Francisco, California.

[Khanna 1992] Sandeep Khanna, Michael Sebrée, John Zolnowsky, *Realtime Scheduling in SunOS 5.0*, USENIX, Winter 1992, San Francisco, California. This describes the real-time features and considerations in this kernel.

[Korty 1989] Joe Korty, *Sema: a Lint-Like Tool for Analyzing Semaphore Usage in a Multithreaded UNIX Kernel,* USENIX Winter 1989, San Diego, California. This describes a tool for doing static lock hierarchy analysis.

[Leffler 1989] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley, Reading, MA, 1989.

[Peacock 1992] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang and Wilfred Yu, *Experiences from Multithreading System V Release 4,* Symposium on Experiences with Distributed & Multiprocessor Systems (SEDMS) III, March 1992, Newport Beach, California.

[Powell 1991] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, *SunOS Multi-thread Architecture,* USENIX Winter 1991, Dallas, Texas. This describes the architecture for user-level multi-threading.

[Stein 1992] D. Stein, D. Shah, *Implementing Lightweight Threads,* USENIX Summer 1992, San Antonio, Texas, pp. 1-10. This describes the implementation of the user-level threads package.

Author Information

Joseph Eykholt is a Senior Staff Engineer and technical leader in the OS-Multithreading group at SunSoft. He received an MSEE from Purdue University in 1978, and a BSEE from Purdue in 1977. Prior to coming to Sun, he was one of the leading developers of multiprocessing features for the Amdahl UTS system, and a logic designer for the Amdahl 580 CPU. His address is SunSoft, Inc., M/S MTV5-40, 2550 Garcia Avenue, Mountain View, CA, 94043. His E-mail address is jre@Eng.Sun.COM. By phone: (415) 336-1849.

Steve Kleiman is a Distinguished Engineer in the Operating Systems Technology Department of SunSoft. He is currently architect of Multi-threading in SunOS. He received an M.S. in Electrical Engineering from Stanford University in 1978 and a B.S. in Electrical Engineering and Computer Science from M.I.T in 1977. He has been involved with the design and development of UNIX and workstation architecture since 1977; first at Bell Telephone Laboratories and then at Sun. He was one of the developers of NFS, Vnodes, and of the original port of SunOS to SPARC. His E-mail address is srk@Eng.Sun.COM. By phone: (415) 336-7295.

Steve Barton graduated from the University of California, Santa Cruz in 1982 with a BA in Computer and Information Sciences. Since then he has worked at Zilog Inc., Parallel Computers, CounterPoint Computers, and Telestream Corp. He is currently a Member of Technical Staff at Sunsoft. He's been with Sun for the last four years. Reach him electronically at steve.barton@Eng.Sun.COM.

Roger Faulkner is a Senior Staff Engineer in the OS-Multithreading group at SunSoft. He received a B.S. in Physics from N. C. State University in 1963 and a Ph.D. in Physics from Princeton University in 1967, then joined Bell Laboratories, where he was seduced by computers. He has been actively involved in the inner workings of the UNIX kernel since 1976 and has done compiler and debugger development along the way. He is one of the principals involved in the development of the /proc file system for SVR4. His E-mail address is raf@Eng.Sun.COM. By phone: (415) 336-1115.

Anil Shivalingiah is a Staff Engineer in the OS-Virtual Memory group at SunSoft. He received an M.S. in Computer Science from University of Texas, Arlington in 1983 and a B.S. in Electronics Engineering from UVCE, India in 1981. He's been with Sun for the last three years. His E-mail address is ans@Eng.Sun.COM.

Mark Smith graduated with a B.S. in Computer Science from the University of California, Santa Barbara in 1986. He is currently a Member of Technical Staff at SunSoft in the OS-Multithreading group. Prior to coming to Sun he worked in the Design Automation department of Amdahl Corp. He can be reached by E-mail at mds@Eng.Sun.COM.

Dan Stein is currently a Member of Technical Staff at Sunsoft where he is one of the developers of the SunOS Multi-thread Architecture. He graduated from the University of Wisconsin in 1981 with a BS in Computer Science.

Jim Voll works in the OS-Multithreading group at SunSoft. He received his B.S. from the University of California, Santa Barbara in 1981. Prior to working at Sun he has worked at Amdahl and Cygnet Systems. He routinely destroys his home directory. His E-mail address is jjv@Eng.Sun.COM.

Mary Weeks has been a member of technical staff at Sun Microsystem since 1986. Prior to Sun, she worked at Xerox. She received her B.A. in computer science from the University of California at Berkeley in 1984.

Dock Williams is a Staff Engineer in the OS-Multithreading group at SunSoft. He received a S.B. in Electrical Engineering and Computer Science from M.I.T. in 1980. He has been with Sun for over six years. Prior to joining Sun, he worked at American Information Systems, ONYX Systems, Tri-Comp Systems, and Hughes Aircraft Radar Systems. His E-mail address is dock@Eng.Sun.COM, phone: (415) 336-1246.

File System Multithreading in System V Release 4 MP

J. Kent Peacock – Intel Multiprocessor Consortium

ABSTRACT

An Intel-sponsored Consortium of computer companies has developed a multiprocessor version of System V Release 4 (SVR4MP) which has been released by UNIX System Laboratories. The Consortium's goal was to add fine-grained locking to SVR4 with minimal change to the kernel, and with complete backward compatibility for user programs. To do this, a locking strategy was developed which complemented, rather than replaced, existing UNIX synchronization mechanisms.

To multithread the file systems, some general locking strategies were developed and applied to the generic Virtual File System (VFS) and vnode interfaces. Of particular interest were the disk-based *S5* and *UFS* file system types, especially with respect to their scalability. Contention points were found and successively eliminated to the point where the file systems were found to be disk-bound. In particular, several file system caches were restructured using a low-contention, highly-scalable approach called a *Software Set-Associative Cache.* This technique reduced the measured locking contention of each of these caches from the 10-15% range to less than 0.1%.

A number of experimental changes to disk queue sorting algorithms were attempted to reduce the disk bottleneck, with limited success. However, these experiments provided the following insight into the need for balance between I/O and CPU utilization in the system: that attempting to increase CPU utilization to show higher parallelism could actually lower system throughput.

Using the *GAEDE* benchmark with a sufficient number of disks configured, the kernel was found to obtain throughput scalability of 88% of the theoretical maximum on 5 processors.

Introduction

The goal of the Consortium assembled by Intel was to produce a multiprocessor version of System V Release 4 in as short a time as possible. As such, there was no inclination to perform a radical restructuring of the kernel, nor to support user-level threads while the evolution of a consensus threads standard was incomplete.

There were two main performance goals for this effort: binary performance running the *GAEDE* benchmark [7] with respect to the uniprocessor system should degrade by no more than 5%; and the proportional increase of throughput should be at least 85% of the first processor for each processor brought online, up to 6 processors. The performance numbers were arrived at through negotiations with UNIX System Laboratories as their acceptance criteria. The 6 processor limit arose from considerations of how well the implementation was expected to scale, namely between 8 and 16 processors, and, more importantly, of how large a system was likely to be available for testing.

Previous Work

Many attempts have been made to adapt UNIX to run on multiprocessor machines. Companies such as AT&T, Encore, Sequent, NCR, DEC, Silicon Graphics, Solbourne and Corollary have all offered multiprocessor UNIX systems, though not all have described the changes made to the operating system in the literature. Bach [2] describes a multiprocessor version of System V released by AT&T. Encore has described several generations of their multithreading effort, first on MACH and then on OSF/1 in a series of papers [4, 11, 12, 15]. This work has the greatest similarity to that reported here, so an attempt has been made to make relevant comparisons to it throughout the paper. DEC has also published papers on their approach to multithreading their BSD-derived ULTRIX system [10, 21]. Ruane's paper on Amdahl's UTS multiprocessing kernel is the best precedent to the philosophy of the approach used by the Consortium [20]. Lately, NCR has discussed their parallelization efforts on System V Release 4 [5, 6]. As NCR has participated as a member of the Intel Consortium, this work influenced the Consortium's approach.

Locking Model

Multiprocessor locking implementations can be divided roughly into two camps: those who replaced the traditional *sleep-wakeup* UNIX synchronization with semaphores or other locking primitives, and those who opted to retain the *sleep-wakeup* synchronization model and add mutual exclusion around the appropriate critical sections. Bach's [2] implementation is of the first type, with existing synchronization mechanisms replaced by Dijkstra semaphores. Ruane's paper [20] represents a sort of canonical description of the type of locking described most often by previous multithreaders using the mutual exclusion approach. The paper gives a very good discussion of some subtle synchronization issues relative to a pre-SVR4 environment, as well as some sound arguments against using Dijkstra semaphores.

Although a thorough description of the Consortium locking protocols is beyond the scope of this paper, there are a number of key features of the locking primitives which should be noted: Firstly, mutual exclusion locks are implemented so that any mutex locks held by a process are automatically released and reacquired across a context switch. This is an imitation of the implicit uniprocessor locking achieved by holding the processor in a non-preemptive kernel. It is necessary for proper *sleep-wakeup* operation for a lock protecting a sleep condition to be released after the sleeping process has established itself on the sleep queue. Most previous efforts have enhanced the *sleep* function to release a single mutex lock, usually specified as an extra argument to the sleep call. With automatic releasing of locks across context switches, sleep calls need not be changed. This means that much of the multithreading effort merely involves adding mutex locks around sections of code, even though they may sleep. Comparisons of many multithreaded files with the originals show this to be literally the only change required. This is an important feature when it is necessary to occasionally upgrade to ongoing releases of the underlying uniprocessor system.

The ability to release all of the locks acquired in the call stack relates to another feature of the locks, namely that recursive locking of each individual lock is allowed. SVR4 is designed in such a way that there are a number of object-oriented interfaces between sub-systems of the kernel, derived from SunOS [8]. These subsystems call back and forth to one another and create surprisingly deep recursive call stacks. (This happens particularly between virtual memory and file system modules.) Though providing considerable modularity, this feature makes it very difficult to establish assertions about which mutex locks might be held coming into any given kernel function. Allowing lock recursion and the automatic release of mutex locks allows a given function to deal only with its own locking requirements, without having to worry about which locks its callers hold, aside from deadlock considerations.

The primitives were also designed to be able to configure whether the caller spins or gives up the processor when the lock is not available. Locks acquired by interrupt routines must spin, whereas locks of the same class which might deadlock one another can avoid deadlock by sleeping. The deadlock avoidance comes from the fact that held locks are released when a lock requester sleeps. In addition, mutex locks may be configured as shared/exclusive locks, allowing multiple reader locks or a single writer lock to be held.

For purposes of the following discussion, it is useful to carefully define the difference between a *resource lock* and a *mutex lock*. Logically, a *resource lock* is a lock which protects a resource even when the locker is not running on a CPU. A *mutex lock*, on the other hand, only protects a resource when the locker is running on a CPU. Stated another way, a resource lock may be held across context switches, while a mutex lock would not be. A resource lock can be an actual locking primitive, such as a semaphore [2], but need not be. Implementing a resource lock as locking data protected by a mutex has been shown by Ruane to provide greater flexibility in locking than the semaphore approach [20]. It should be noted that there are resource locks already present in the uniprocessor code, for example, the B_BUSY flag bit in a disk buffer cache header or the ILOCKED flag in an inode structure. Protecting manipulations of these resource locks with mutexes is usually sufficient to generalize them to work on a multiprocessor. In fact, all of the instances of a given type of resource lock can be protected using a single mutex lock rather than a lock per instance, typically with very low contention on the mutex.

More detailed descriptions of the locking primitives and more detailed arguments in favor of using them over semaphores can be found elsewhere [5, 17].

File Systems Locking

The SVR4 file system implementation centers around two separate object-oriented interfaces which originated in SunOS [9]. One is the Virtual File System (VFS) interface, which consists of functions which perform file system operations, for example, mounting and unmounting. The other is the virtual node (vnode) interface, which allows operations on instances of files which reside in a virtual file system.

Most of the file system multithreading effort was spent developing a general multithreading model for these two interfaces. Although there are around 10 file system types in SVR4, only the two disk-based file systems are discussed: the UNIX File System (UFS) type, which is a derivative of the Berkeley Fast File System [13] via SunOS, and the S5 type, which is derived from the System V Release 3 file system. The strategies for multithreading these two file systems were basically the same.

VFS Locking

The VFS layer of the file system provides the support for an object-oriented interface to the various file system types available. This support includes maintaining the list of mounted file systems and resolving races between file system unmounts and pathname searches into a file system being unmounted. The multithreading of the VFS layer centers around a shared/exclusive mutual exclusion lock which serializes access to mount points during pathname lookup, mount and unmount operations. This lock protects the existing uniprocessor resource lock on each virtual file system. This approach seems to be essentially the same as that used by Encore in their multithreading of the Mach vnode-based file system [11].

Vnode Locking

The vnode layer of the kernel implements another object-oriented interface for operations on each file within a virtual file system of a given type. The focal point of these operations is the *vnode* structure, which is usually contained within a VFS-dependent node structure for each active vfs element in the system. The object-oriented interface consists of a number of macros prefixed with ''VOP_'' which call through a VFS-dependent function code table. With only a few exceptions, these macro calls represent the only path into the file system code. The interface was designed to support a set of operations which were atomic and mostly stateless, in order to support a stateless network file system, namely NFS [9]. In actual implementation, local file systems do retain some state necessary to implement normal UNIX file system semantics. For example, in a UFS or S5 filesystem, a VOP_LOOKUP operation, which does one stage of a pathname lookup, leaves the found file or directory in a locked state on return.

The granularity of locking desired at the vnode level is the individual vnode, which suggests a locking strategy whereby a lock on a vnode is obtained and released around almost every VOP call. Using this strategy provides a blanket level of essentially automatic vnode locking for most of the VOP functions.

Inode Locking

There are two main aspects to file system locking inside UFS and S5: locking of a given file inode while some operation is performed on it, and the locking of the file system inode cache during the lookup or freeing of an inode. Inode cache locking is discussed in a later section devoted to cache contention.

The per-inode lock is a resource lock which can remain held across blocking operations, such as disk I/O. The original uniprocessor implementation of the inode locking uses two separate lock flag bits in the inode, ILOCKED and IRWLOCKED. (Pre-SVR4 systems had only the ILOCKED flag.) The ILOCKED flag locks the inode during most operations which would access or change the contents of the inode itself, whereas the IRWLOCKED flag makes read or write operations atomic. This allows a long read or write operation to proceed without blocking a *stat* operation, for example. These locks are set and cleared by the ILOCK – IUNLOCK and IRWLOCK – IRWUNLOCK macro pairs. Both the ILOCKED and IRWLOCKED bits can be held at the same time by two different processes. In a multiprocessor, the processes must be prevented from running at the same time for correct emulation of the uniprocessor semantics. In order to accomplish this, the vnode lock must be held concurrently with each of the two resource lock flags. This is done by acquiring the vnode lock from within ILOCK and IRWLOCK and releasing it within IUNLOCK and IRWUNLOCK. The effect of this on return from a VOP function which does an ILOCK without an IUNLOCK, due to lock recursion, is to leave the vnode locked outside the VOP function. For example, the previously mentioned VOP_LOOKUP function returns with both ILOCKED set and the vnode lock held on the found directory or file. If the process does a context switch after the return, the vnode lock is released, but the resource lock, embodied in the flag bit, is not. Hence, the integrity of the inode is still protected, although a process holding the IRWLOCKED lock could run after acquiring the vnode lock. This behavior is the same as in the uniprocessor system. When the next VOP call is made which does the matching IUNLOCK on the inode, the extra vnode lock on the file is released, and the vnode is unlocked completely on return from the call.

This locking approach represents a different philosophy from the OSF/1 file system [12], where by design, no file system locks can be held on return from a VOP function. Since the uniprocessor model is preserved by our strategy, no additional races are introduced such that an error path could cause the lock not to be released. The only way for the vnode lock not to be released is for the inode lock not to be released, which would also qualify as a uniprocessor bug. Having the vnode lock visible at the VOP interface also allows the lock to be held around multiple VOP calls, which is made possible by the recursive property of the Consortium mutex locks. This is used in several cases to avoid a race with file locking on a vnode.

Because there is a lock on each vnode, the potential for deadlock exists when it is necessary to lock more than one vnode, as during pathname searches or some STREAMS operations. The solution to this problem is to have the lock *sleep* when it waits for a busy vnode lock. As previously discussed, this removes the deadlock possibility because all of the locks held by the locker are released. The one problem that this causes is that potential context-switches are introduced which were not present before the vnode locking was added. One solution to this is to widen the scope of the vnode locking so that the preemption happens in a safe place. Locking around the VOP calls has been found to be sufficiently wide for almost all problem situations, since callers should conservatively assume that any VOP call might sleep. Additional vnode locking which is bound with inode lock manipulations is also safe, because the inode locking itself may sleep. The only place where the vnode lock's possible sleeping causes a problem is in the *iget* function where the inode has been found by a cache lookup. A sleep to wait for the vnode lock could allow the identity of the inode to change. The solution is simply to recheck after the vnode is locked that the inode obtained still matches the identity of the one searched for.

When an inode is heavily shared by many processes, the processes tend to queue up sleeping to wait for the inode lock. This represents an example of what has been called the *thundering herd* problem [5]. When an inode lock is released, the normal *wakeup* function makes all of the processes sleeping on the inode runnable. In a multiprocessor, this results in all but one of the processes finding the lock busy and going back to sleep. This takes $O(N^2)$ CPU time to process N sleepers. To alleviate this, a *wake1proc* version of *wakeup* has been used to awaken only one process when an inode lock is released. The process awakened using this approach has to assume more responsibility: if the process no longer wants the inode lock, it must do another *wake1proc* to pass the lock to another waiting process.

VN_HOLD/VN_RELE Locking

Vnodes represent shared resources in most cases, and can hence be referenced by a number of different processes in a completely asynchronous fashion. When a new pointer reference to a vnode is created, a reference count in the vnode is incremented by doing a VN_HOLD (which uses an atomic add operation in the multiprocessor case). When the caller is done with the vnode, it releases the reference by doing a VN_RELE operation. The VN_RELE does an atomic decrement of the reference count and if it becomes zero, calls VOP_INACTIVE, which does a file system dependent cleanup operation on the vnode.

In some file systems the vnode continues to exist in a quiescent state after the VOP_INACTIVE, such that it can be reclaimed by a future lookup operation. The problem with this is that the zero count state is used to signify the quiescent state in these file systems. But the count goes to zero before the inactivation is performed, so there is a serious race between the lookup operation and the inactivation. It is possible for another process to find the vnode and do a VN_HOLD followed by a VN_RELE before the inactivation is performed, resulting in two inactivation attempts. To solve this problem it is necessary to change the file systems not to use the zero count as the inactive indicator. For example, in the S5 and UFS file systems, the count is decremented again to –1 under the inode cache lookup lock to *truly* indicate the inactive state. On a lookup, which is also done holding the cache lookup lock, when the count for an inode is –1, the inode is reactivated and the count set back to zero before doing a VN_HOLD to count the new reference. This solution makes the state transitions between active and inactive states safe, while allowing the use of atomic operations for VN_HOLD and VN_RELE.

Encore describes using a per-vnode spin lock around the VN_HOLD/VN_RELE increment and decrement operations [11], but they do not reveal their solution to the above race condition.

Performance Tuning

A number of useful tools were available to the SVR4MP developers [6, 17]. These tools, together with techniques very similar to those described by Paciorek, *et al.* [15], allowed reasonably accurate characterization of the locking contention and scalability of the system and the detection of deadlocks.

Cache Contention

A number of different caches are used within all UNIX variants to enhance system performance. Logically, such caches are collections of objects each tagged with an identifier. Typical operations perform some actions on an individual cache object. These actions can often be quite self-contained and very scalable, since their locality is constrained. However, once the operation enters the domain of the cache itself, it can collide with other operations on different objects in the cache. Hence, attention is naturally drawn to these caches as places where processor contention can occur and needs to be relieved. Most of these caches have a similar structure, as illustrated in Figure 1. In Figure 1, the hash queues are an array of queue head structures, indexed by computing a hash function of an object identifier during a cache operation. Each square box represents an element of the cache, and those that are not marked "Busy" are chained in least-recently-used (LRU) order on a free list. To aid in discussing locking strategies for this organization it is useful to consider the three main operations: lookup, release and flush.

A cache lookup operation scans the appropriate hash queue for the given object identifier. If an element matching the identifier is found and is not busy, it is removed from the free list, marked busy and returned to the caller. If the matching element is busy, then the caller waits for it to be released by the current owner and then rescans the hash queue to make sure the identity of the matched element has not changed. If the identifier is not found, an element is removed from the front of the free list and its current hash list and replaced on the hash queue of the searched-for identifier. The element is marked busy, its contents are then changed to reflect the new identity and it is returned to the caller. The release operation of a busy cache element puts a busy element back onto the free list and clears the busy mark.

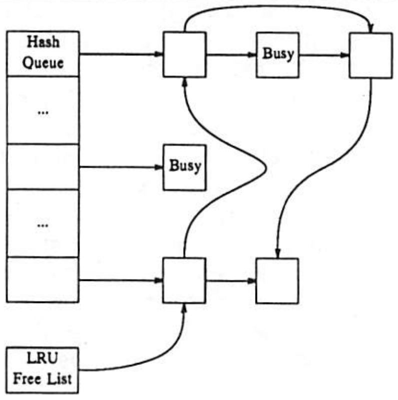


Figure 1: General Cache Organization

The flush operation runs periodically to clean elements on the free list so that they can be reused immediately when a cache miss occurs. The flush operation scans the list for items that need cleaning, removes them from the free list, cleans them and returns them to the free list when done. As an example, the buffer cache cleaning involves queueing modified disk blocks to be written to the disk. A very detailed description of this structure relative to the disk buffer cache can be found in Bach [1].

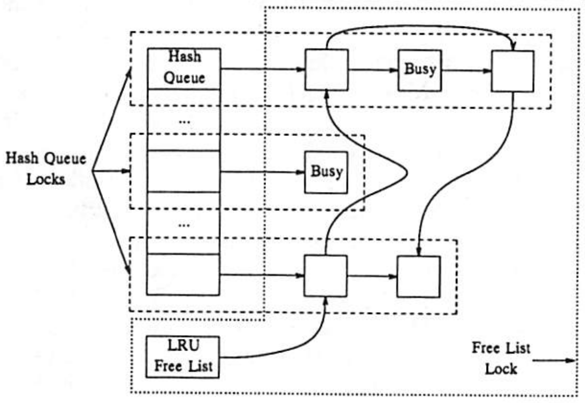


Figure 2: Separate Hash Queue and Free List Locks

The "intuitively obvious" way to lock this structure involves the use of three types of locks: a lock on each cache element, a lock for each hash queue and a lock to protect the free list, as shown in Figure 2. The dotted and dashed lines enclose the data structures protected by each of the hash queue and free list locks (the cache element locks are not shown). This locking strategy was used by Encore in their Mach/4.3BSD buffer cache locking, and has a number of deadlock and performance difficulties [4]. In terms of overhead, this locking requires obtaining at least three locks per lookup or release operation, including the per-element lock. A cache lookup miss may require obtaining two arbitrary hash queue locks, thus requiring a deadlock avoidance strategy to be implemented. More importantly, the free list lock is obtained for every operation, so it represents the limiting factor for the scalability of the cache, as discovered and reported by Encore.

A simplification of this approach lowers the locking overhead without sacrificing scalability (assuming short average hash queue size) by using only the single free list lock to protect all of the data structures, as shown in Figure 3. An early version of the Consortium locking for the buffer cache included a per-element lock along with the free list lock. It was noted that an element lock was almost always obtained while holding the free list lock, and was hence redundant. Removing the element lock reduced lock overhead by 50% and lowered contention also. Of course, the element lock can not be removed if it is a semaphore resource lock which replaces the busy mark on the cache element, as in the semaphore locking approach. In the Consortium case, the element lock was a mutex which only protected manipulations of fields within the data structure.

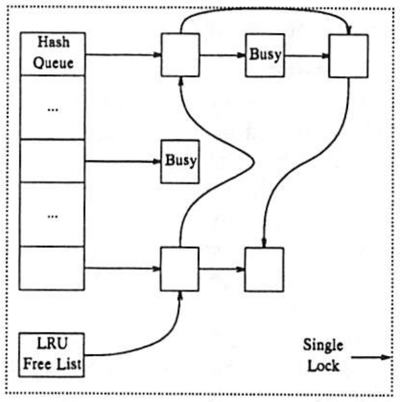


Figure 3: Single Lock for All Cache Data Structures

Unfortunately, the single lock still represents a contention point when the access rate to a cache is high. A solution to this problem is to fragment the free list into segments which are associated with each hash list and use a lock on each hash list to protect both the hash queue and the free list, as shown in Figure 4. Any time a free element is required, it is allocated from the free list associated with the hash queue being searched, so that each cache element is permanently bound to a fixed hash queue. Because of this, each hash queue is typically initialized to have a configurable, constant number of elements. This cache organization was dubbed a

Software Set-Associative Cache (SSAC) due to its logical resemblance to a hardware set-associative cache, and is described in detail elsewhere [17]. It represents the key technique used in multithreading the file system caches to obtain practically unlimited scalability. The description is extended here to show the application of the technique to a number of different caches and illustrate the problems that arose.

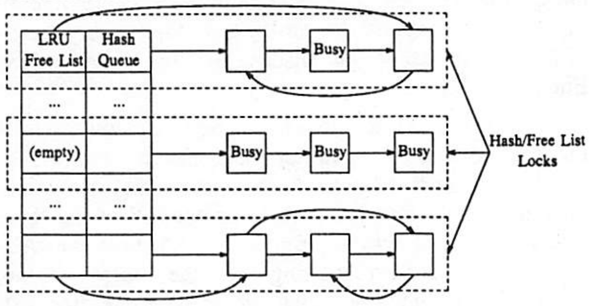


Figure 4: Software Set-Associative Cache Locking

A follow-on paper by Encore describing parallelization of a vnode-based file system [11] does not refer to the free list contention problem. The buffer cache locking is described as consisting of a lock on each hash queue, plus a lock inside each buffer. From this description it is not possible to determine how or even if they have solved the free list contention problem. The problem is likely moot to them, however, as their stated intention was ultimately to replace the buffer cache entirely with the Mach "No Buffer Cache" code from CMU.

File System Caches

While tuning the file systems, three SVR4 file system caches showed up as having significant levels of cache contention. The first of these caches is the *segmap* cache, which is used to map file pages into windows where they can be copied during read and write operations. The segmap cache replaces the buffer cache in SVR4 for most data read and write operations. The second cache is the buffer cache (a shadow of its former self), which is used to hold file system structural information, such as blocks of inodes. The last cache is the directory name lookup cache (DNLC), which is a cache of pathname component translations. The segmap cache exhibited 10-15% contention with only two processors running cache-bound file system I/O, while the buffer and DNLC caches had similar contention when the GAEDE and AIM 3 benchmarks were run on 5 CPUs. The SSAC technique was applied to these caches, with slightly different structures in each case. The locking contention for all of the caches was reduced to less than .1%. The different structures lead to some small problems which highlight the properties of the approach.

Segmap Cache

The *segmap* cache was changed first, as it represented the largest bottleneck. In this case, the hash queue and free list pointers in the cache structure were kept. The hash and free queues for each hash queue were set up at initialization to contain 4 cache elements each. The hash queue manipulations were simplified, since elements are never moved from one hash chain to another. This code change was very straightforward and took less than one day to accomplish. No side effects or difficulties were apparent after the change, and the contention was reduced as described. Segmap cache elements are not locked for exclusive use on return from the lookup operation, but a reference count is incremented. Hence, more than one process may actually use a cache entry. To protect these processes from one another, the hash queue lock for the queue containing an element is locked while fields in the cache element are manipulated. This avoids the necessity for a separate mutex lock to protect the data contents within the cache. Contention for this per-element locking is included in the less than .1% contention measurement.

Disk Buffer Cache

The application of the SSAC structure to the disk buffer cache proved to be somewhat more interesting and challenging due to some features of the SVR4 buffer cache code. Memory for the cache headers and buffers is not statically allocated at initialization, as it was in SVR3. Rather, the free list of headers grows as needed and the buffers themselves are dynamically allocated and freed to accommodate different buffer sizes. This made it difficult to configure a set of fixed-length buffer chains on the hash queues, so another approach was needed. In addition, the cache hashing function was found to be very poor. Most kernel hash queue arrays are configured to have 2^n entries so that the cheaper bitmask operation (tag & (2^n-1)) can be used instead of the modulus operation to fold the object identifier tag into a hash index. If the object identifiers are separated by a power of two, then only a subset of the hash queues are actually ever used. In the case of the buffer cache, a UFS file system with 4-Kbyte blocks generates object identifiers which are usually multiples of 8 (when converted to 512-byte physical block numbers), thus using only every 8th hash queue. This problem can be alleviated by using only 2^n-1 entries, which requires the more expensive modulus operation, but gives a better hashing distribution. (As another example of this, process structures were always allocated on 512-byte boundaries, so any *sleep* on the address of any proc structure always queued the sleeper on sleep hash queue 0.)

One of the desirable properties of a cache is that the hash queue remain fairly short, to reduce the search time required for a lookup. A traditional target has been to keep the average size of each hash queue at around 4 buffers. When the free list for each hash queue was configured to be this size, it was discovered that the buffer cache code contains an inherent deadlock. Certain operations require the use of 2 or more buffers, which can deadlock when all of the buffers on a free list are used up by such operations. This deadlock could not be eliminated without radical restructuring of the code. So, to make its occurrence less likely, larger free lists were configured and shared among a number of hash queues, with a single mutex lock per free list. Each hash queue is bound at initialization to a particular free list, and buffers do move between hash queues, but only within those bound to the same free list. This approach provided the contention reduction desired, while increasing the probability of the deadlock only marginally.

Directory Name Lookup Cache

The DNLC cache changes were another variation on the SSAC theme. All of the free list and hash queue pointers were removed and the cache structured as a 2-dimensional array, with a vector of simple spin locks to lock each row of the cache. To maintain the LRU ordering of each cache line, a timestamp was added to each DNLC structure indicating the last time that the structure was accessed, with a 0 timestamp indicating a busy cache entry. When a cache search is attempted, the entry with the lowest non-zero timestamp is remembered and reused if the name is entered into the cache.

The semantics of this particular cache have some unusual features, most of which do not affect the SSAC structure. The lookup and enter operations are split into two distinct operations, where they are normally combined. A number of operations purge entries from the cache based on different characteristics, such as vnode identity, file system identity, just any old entry or the entire cache. The purge operation actually removes the entry, rather than cleaning it as in other flush-type operations. Because of this, a little more care needs to be taken in deciding which entries should be purged when merely reclaiming space. In most of the purge operations, the entire cache has to be scanned and all elements satisfying the search criterion must be removed. The division of free list locks in this case is an advantage, because each row of the cache is locked individually as the cache is searched, allowing other operations to proceed in parallel. On the other hand, the multiple locks do not allow the state of the entire cache to remain frozen during an operation. (Although this property does not present a problem in normal operation, a debugging function which counts all of the occurrences of a given vnode in the cache is no longer reliable.)

Since there is no single LRU free list, purging an element when any one will do has to be done carefully. The approach of traversing each cache row processing each element in turn works very well in the cleaning case (for example, the buffer cache), where the cleaned element is not reused immediately. In the purging case, it is a poor approximation to selecting a least-recently-used entry to purge. A better LRU approximation for the purging case is to select the least-recently-used entry from a cache row, purge it and then move onto the next row in the cache, cycling through the cache.

A potential problem was noticed in the hashing function for this cache. The same entry can be entered into the cache with different user permissions (credentials), so an often-used directory entry could be in the cache many times with different credentials for different users, all hashed onto the same queue. This could cause thrashing on that queue. The solution to the problem is to use the credential pointer in the hash calculation to distribute the occurrences of the name to a number of different hash queues. This illustrates a general property of the SSAC organization, which is that the goodness of the hashing function becomes more important when relatively short, fixed-sized cache rows are defined. In general, the shorter each row is, the faster the search can be, but the probability of thrashing within a row increases. Experience from hardware caches, where 4-way set associativity is considered adequate, suggests that 4 is a reasonable starting value for tuning the hash queue length.

The DNLC cache also has some other interesting behavior in its interactions with the UFS and S5 file systems. When a vnode is entered into the DNLC cache, it has a VN_HOLD placed on it. Quite often, the last reference to a UFS or S5 file is from the DNLC cache. Because of this, the *iget* function, which attempts to find a given file in the inode cache, calls the DNLC purge-any function when there are no more free inodes in the cache. This purging represents a hit-or-miss effort which is repeated until one of the file system's inodes whose last reference is from the DNLC cache is purged. If one is hit, the purge function calls VN_RELE to decrement the vnode reference count to zero, which then calls VOP_INACTIVE to release its inode. In the S5 file system, this is where the fun began: If the file being inactived was unlinked, then the inactivate routine called the *ifree* function to release the inode number back to the inode free list. If the call to *iget* was for a file being allocated (from *ialloc*), then the *ifree* function would deadlock trying to obtain the non-recursive mutex lock on the file system structure, which was held by *ialloc*. The solution to the problem was to release the mutex lock in *ialloc* before the call to *iget*. Unfortunately, this allowed a race where attempts could be made to allocate the same inode more than once, which had to be dealt with by detecting the already-allocated inode after return from *iget*. This example is cited to illustrate the (often unexpected) recursive flavor of the SVR4 file system code.

The Encore approach to locking the DNLC cache is the combination of hash and free list locks shown in Figure 2 [11]. As such, each operation acquires between 1 and 4 locks to accomplish an enter operation, whereas the SSAC approach always acquires only 1 row lock with lower contention than in the Encore case due to the fragmented free lists.

Inode Caches

The inode caches in the UFS and S5 file systems are additional candidates for the SSAC multithreading approach. However, a single lock has been used for the entire collection of inode hash queues and the inode free list in each file system. This strategy was chosen due to an insufficient level of measured contention. It could be argued that the lack of contention arises from the relative infrequency of file open or create operations relative to others, such as reading or writing. Also, the DNLC cache removes some of the inode lookup load caused by pathname searches. The Encore locking approach is the same as that used for their other caches, namely the use of a lock for each hash queue as well as a free list lock.

Disk Queue Tuning

Once the contention on the three caches was removed, a 2-CPU system with a single disk became essentially I/O-bound running the GAEDE benchmark. To investigate the cause of this, some performance measuring was added to the kernel which reported utilizations of all processors and the disk controller, as well as the size of the disk queue. These measurements were collected at each clock tick and averaged over a 1-second interval. Watching these statistics in real time revealed that the system was alternating between periods of processor saturation with little or no disk activity, and disk saturation with little or no processor activity. The disk saturation occurred when the disk queue size rose to several *hundred* requests in length. Unfortunately, it is theoretically possible in SVR4 for the size of the disk queue to be bounded only by the number of allocatable memory pages in the system.

The blocking of the CPUs during periods when the disk queues were very large was due to the fact that processes were blocked on synchronous I/O which was generated to update inode contents and directories. As an experiment, all of the synchronous inode update operations were changed to delayed writes to decouple them from the disk driver. This change improved the CPU scalability substantially, at the cost of a less consistent file system. On a two-processor system, the GAEDE benchmark could be transformed from mostly disk-bound to mostly CPU-bound with this change. (A kernel variable which turned this feature on or off was cynically dubbed the "benchmark flag", with the suggestion that at least some vendors actually have such a flag.) McVoy [14] proposed doing something similar to this, but with better file system consistency, by forcing the correct ordering of asynchronous operations to the disk for directory and inode updates. The change described here was not intended primarily as a file system enhancement, but rather as a tool to make the benchmark more CPU-bound to uncover locking contention in the file systems.

Another approach, which has been used successfully to increase write throughput of the buffer cache in SVR3 [16], was to queue synchronous requests near the front of the disk queue. This change gave only minimal benefit. It was found that requests were often queued asynchronously, but then a process would later decide that it wanted the data block. This suggested an approach whereby requests which were being waited for were dynamically promoted to the front of the disk queue, to allow waiting processes to become active sooner. Surprisingly, this change caused overall throughput of the benchmark to decline dramatically, because the disk queue never actually became empty, but increased monotonically in size.

It appears from this that balance among utilizations of resources is more important than optimizing the utilization of one class of resource, namely the CPUs. The segments of idle CPU time seem to be necessary to restore balance by lowering the effective request rate to the disk, allowing it to clear the accumulated backlog.

The real problem here is that there is a "high wall" between the disk driver and upper level file system layer. Once an asynchronous write request is passed to the disk driver, there is no current way to retrieve it until the operation is complete. This represents wasted I/O activity, since a process that waits for such a request is likely to modify the data just written. Similarly, the disk can be idle for long periods of time because the cache flushing code which could give the driver some work to do does not know that the disk is idle. When it does give it work to do, it tends to flood the disk queue with requests, which overloads the driver and slows or idles the CPUs.

Fixing these problems to even out the flow of cache copy-back traffic to the disk is one part of the solution to this problem. The other part of the solution is to reduce the write requirements of the file systems themselves, possibly by the use of log-structured file systems such as that implemented for the Sprite project [19].

Related Work

The closest work to this effort was done by Encore to multithread their vnode-based Mach file system and the OSF/1 file system [11, 12]. Their approach was actually quite different in a number of ways.

The most significant difference is that they chose to lock the file systems below the vnode layer, that is, there is no generic locking outside the file system. The SVR4MP approach was the exact opposite of this. The locking is provided in a generic sense across the vnode operation interface, with the vnode lock available as the mutual exclusion lock for all of the per-vnode data, including file system dependent data. The Encore approach replaced the inode ILOCKED flag with read/write resource locks, which extended the file semantics to allow true concurrency for file reading. The rationale for this extension was that some files are read frequently on a large system, and that performance could be enhanced by reducing lock contention on these files. During Consortium tuning, this type of contention was not encountered, probably due to our configurations being smaller in size than Encore's. Another advantage of locking below the vnode interface is that the file system writer has the flexibility to determine whether a read/write or simple mutex lock should be used, whereas the Consortium approach allows only mutex locking. On the other hand, the Consortium approach involves less semantic (i.e. code) changes, particularly since it is permissible for a vnode lock to be held around more than one VOP call when that is required. This was important due to the time-to-market and robustness constraints that the project faced.

Encore avoids contention by releasing the resource locks across any blocking operations, which allows a race between two simultaneous lookups of the same inode. This race requires a recheck of the hash queue whenever a new inode is read from disk. In the OSF/1 file system, the recheck is avoided by timestamping data structures, such as a hash queue, when they change and only rechecking a queue if its timestamp changes across a blocking operation. Much of the paper on OSF/1 describes the races introduced into directory operations by more permissive locking and how they were solved using timestamps. In the Consortium approach, the original resource locking semantics are preserved, keeping the ILOCKED or IRWLOCKED flags locked across blocking operations, even though the vnode mutex locks are released.

Another clear difference can be seen in the general cache locking strategy, which has been commented on throughout the paper. Practically all of their described cache locking fits the hash queue lock + free list lock + element lock model, as shown in Figure 2. The Consortium locking uses the single lock model shown in Figure 3, modified to use SSAC locking (Figure 4) where contention is too high.

Summary

The SVR4MP effort resulted in a system which largely met its goals. The degradation relative to a uniprocessor SVR4 kernel is very close to the 5% target, and the throughput scalability at 5 processors is 88% of the theoretical maximum. This is computed as 5-CPU Throughput / (5 * 1-CPU Throughput). Figure 5 is a graph which shows the actual increase in throughput as a function of the number of processors, as well as the ideal scalability. To achieve this goal given the disk-bound behavior described previously, it was necessary to configure 8 separate disk drives on 4 disk controllers in the benchmark system (with the "benchmark flag" turned off).

The file system multithreading effort resulted in a model which provides a lot of generic support for locking within each file system type. The vnode locking around VOP functions provides default locking protection for most file system functions, making individual file system multithreading easier.

Significant contention points in the file system caches were relieved by the application of the Software Set-Associative Cache structure. Once these significant locking bottlenecks were relieved, the file systems were found to be inherently disk bound by our benchmarks. Some disk queueing modifications were tried, with limited success and one dramatic failure. However, these modifications lead to insights about the importance of maintaining balance to achieve optimal throughput, rather than attempting to optimize the utilization of a single resource class.

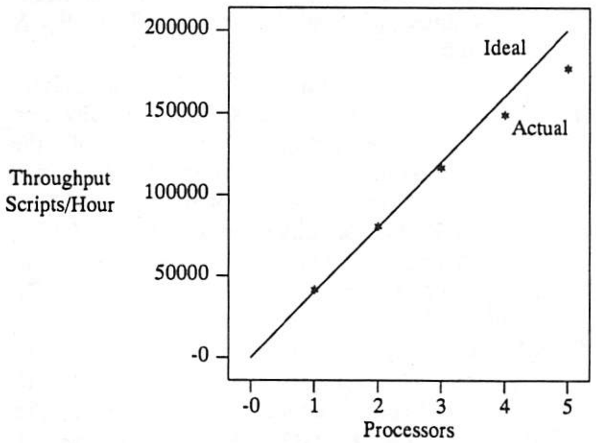


Figure 5: Scalability of the Gaede Benchmark

The Sprite Logging File System is reportedly an order of magnitude more efficient at using disks than existing file systems [19], running at CPU saturation while consuming only 17% of disk bandwidth in a test which is similar to GAEDE. This being the opposite of our situation, the marriage of multiprocessor technology with log-based file systems appears to be quite desirable.

SVR4MP is available from UNIX Systems Laboratories as a source-code upgrade to System V Release 4. More information can be obtained by calling 1-800-828-UNIX.

Acknowledgments

This project has been a pleasure for us to be involved with, as most of the architecture team had at least two generations of multiprocessor design experience. A large number of people have contributed to the success of SVR4MP, most having written code: Mike Abbott, Sunil Bopardikar, Fiorenzo Catteneo, Calvin Chou, Ho Chen, Ben Curry, Jane Ha, Jim Hanko, Mohan Krishnan, John Litvin, Mani Mahalingam, Arun Maheshwari, Cliff Neighbors, Sandeep Nijhawan, Mark Nudelman, Lisa Repka, Sunil Saxena, K. M. Sherif, Moyee Siu, John Slice, Dean Thomas, Vijaya Verma, Fred Yang and Wilfred Yu. Special thanks are due to Wilfred Yu for the performance numbers quoted here. Thanks are also due to the reviewers for their helpful suggestions for improvements to this paper.

References

[1] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs NJ, 1986, pp. 38-59.

[2] M. Bach and S. Buroff, Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 64:1733-1749, October 1984.

[3] R. Barkley and T. P. Lee. A Dynamic File System Inode Allocation and Reclaim Strategy. Froceedings of the Winter 1990 USENIX Conference.

[4] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis. Froceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems, pp 105-126, October 1989. Also appeared as: Mach/4.3BSD: A Conservative Approach to Parallelization. *Computing Systems*, Vol. 3, No. 1, USENIX, Winter 1990.

[5] M. Campbell, Richard Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith and R. Wescott. The Parallelization of UNIX System V Release 4.0. Froceedings of the Winter 1991 USENIX Conference.

[6] M. Campbell, R. Holt and J. Slice. Lock Granularity Tuning Mechanisms in SVR4/MP. Froceedings of the Second Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II). USENIX, March 1991.

[7] S. Gaede. A Scaling Technique for Comparing Interactive System Capacities. Froceedings of the Conference of CMG XIII, December 1982.

[8] R. Gingell, J. Moran and W. Shannon. Virtual Memory Architecture in SunOS. Froceedings of the Summer 1987 USENIX Conference.

[9] S. Kleiman. Vnodes: An Architecture for Multiple File Systems in Sun UNIX. Froceedings of the Summer 1986 USENIX Conference.

[10] G. Hamilton and D. Conde. An Experimental Symmetric Multiprocessor ULTRIX Kernel. Froceedings of the Winter 1988 USENIX Conference.

[11] A. Langerman, J. Boykin and S. LoVerso. A Highly-Parallelized Mach-based Vnode Filesystem. Froceedings of the Winter 1990 USENIX Conference.

[12] S. LoVerso, N. Paciorek, A. Langerman and G. Feinberg. The OSF/1 UNIX Filesystem (UFS). Froceedings of the Winter 1991 USENIX Conference.

[13] M. K. McKusick, W. Joy, S. Leffler and R. Fabry. A Fast File System for UNIX. *Transactions on Computer Systems*, Vol. 2 No. 3, pp 181-197. ACM, 1984.

[14] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. Froceedings of the Winter 1991 USENIX Conference.

[15] N. Paciorek, S. LoVerso, A. Langerman. Debugging Operating System Kernels. Froceedings of the Second Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II). USENIX, March 1991.

[16] K. Peacock. The Counterpoint Fast File System. Froceedings of the 1988 Winter USENIX Conference.

[17] J. K. Peacock, S. Saxena, D. Thomas, F. Yang and W. Yu. Experiences from Multithreading System V Release 4. Froceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III). USENIX, March 1992.

[18] R. Rodriguez, M. Koehler, L. Palmer and R. Palmer. A Dynamic UNIX Operating System. Froceedings of the Summer 1988 USENIX Conference.

[19] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. Froceedings of the Thirteenth ACM Symposium on Operating Systems Principles. ACM SIGOPS, Vol. 25, No. 5, October 1991.

[20] L. M. Ruane. Process Synchronization in the UTS Kernel. *Computing Systems*, Vol. 3 No. 3, USENIX, Summer 1990.

[21] U. Sinkewicz. A Strategy for SMP ULTRIX. Froceedings of the Summer 1988 USENIX Conference.

Author Information

Kent Peacock has worked at Intel as a Consultant for 2 years as a member of the Intel Multiprocessor Consortium. Prior to that, he worked at Acer/Counterpoint developing a multiprocessor implementation of System V and the Acer/Counterpoint Fast File System, which is now part of SCO UNIX [16]. He has worked on design and implementation of 5 multiprocessor systems since 1978, and has dabbled in performance tuning, C compilers, multiprocessor debugging tools and graphics applications. He graduated from the University of Waterloo in Ontario, Canada with a Ph.D. in Computer Science in 1979, having previously completed a Master of Mathematics in Computer Science in 1975. In 1974, he graduated from the University of Manitoba, in Winnipeg, Canada, with a Bachelor of Science in Electrical Engineering. He can be reached via U. S. Mail at 1747 Fanwood Ct.;San Jose, CA 95133 and electronically at kentp@stps18.intel.com.

The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment

Mary Baker, Mark Sullivan – University of California, Berkeley

ABSTRACT

As organizations with high system availability requirements move to UNIX, the elimination of down-time in the UNIX environment becomes a more important issue. Designing for fast recovery, rather than crash prevention, can provide low-cost highly-available systems without sacrificing performance or simplicity. In Sprite, a UNIX-like distributed operating system, we accomplish this fast recovery in part through the use of a *recovery box*: a stable area of memory in which the system stores carefully selected pieces of system state, and from which the system can be regenerated quickly. Error detection using checksums allows the system to revert to its traditional reboot sequence if the recovery box data is corrupted during system failure. Recent statistics about the types and frequencies of operating system failures indicate that fast recovery using the recovery box will be possible most of the time. Using our recovery box implementation, a Sprite file server recovers in 26 seconds and a database manager with ten remote client processes recovers in six seconds – fast enough that many users and applications will not care that the system crashed.

Introduction

Increasing workstation performance is making UNIX and related operating systems more attractive in environments that also value high system availability. Unfortunately, measurements from Internet sites [12] indicate that UNIX machines fail on average once every two weeks. To maintain high availability, these systems must either reduce the failure rate substantially or recover very quickly after errors.

Traditional fault tolerant systems strive for high availability by eliminating or masking failures, so that the system never goes down. In doing so, they sacrifice some combination of the traditional strengths of the UNIX environment: low hardware cost, high performance during normal execution, and simplicity. For example, Tandem [5] and Stratus [23] provide non-stop processing through hardware redundancy and sometimes software redundancy. Auragen [7] applies redundant hardware and a modified process pair scheme to the UNIX environment. The hardware support is costly, and redundant software techniques either reduce normal performance or increase implementation complexity. Harp's [11] application of replicated hardware and software to NFS file servers suffers no performance degradation, but it increases system complexity. HA-NFS [6] improves the availability of NFS file servers through specialized, redundant hardware.

Other fault-tolerant systems provide high availability through recovery schemes that are significantly more complex than the usual UNIX recovery path. Integrity-S2 [10] detects system errors as they occur and attempts to correct affected internal data structures while the system is running. MVS [3] uses a multi-level recovery scheme in which different portions of the system can fail and recover independently. In contrast, UNIX system designers have generally chosen a simple recovery paradigm that is easy to understand and test: when UNIX detects a serious error, the system just shuts down and reboots from scratch (a ''hard reboot'').

To provide relatively high availability while retaining UNIX strengths, we have built a system that allows failures but recovers quickly using a simple, two-tiered recovery mechanism. After a failure, the system first tries to recover quickly from backup data that it stored in main memory during regular execution. If this fast recovery fails, the system just reverts to the traditional disk-based hard reboot. Existing failure statistics [9, 21, 22] show that most common failures will not corrupt the main memory backup data, so the fast recovery path should be successful most of the time. As long as corrupted data is detected, the traditional recovery path will allow us to retain current reliability.

In order to preserve system state across failures, we have designed and implemented a *recovery box* in Sprite [15], a distributed UNIX-compatible operating system. The recovery box is an area of memory used to store recovery information. It can be implemented using non-volatile RAM for protection from power failures. During execution, the operating system stores backup copies of system data in the recovery box. For example, the Sprite file server stores data about files open or cached on its client workstations. After a failure, the system retrieves these items from the box. If any errors are encountered during recovery, the recovery box is cleared and the system reboots from scratch in the traditional fashion. If no errors are detected, the retrieved items are used to regenerate the system state quickly. Computing and storing checksums on items inserted into the recovery box helps detect memory corruption caused by software.

We have provided a data insertion and retrieval interface that allows the recovery box to be used by both the operating system and application programs. The recovery box has been used for fast recovery in the Sprite distributed file system and in an experimental version of POSTGRES [19], a database management system (DBMS) with clients running on different machines. Using the recovery box and other techniques, we have reduced the combined recovery time for a Sprite file server and the POSTGRES DBMS from many minutes to a total of 32 seconds. Other applications could use the recovery box to reduce down time if they have state that is: (1) slow to regenerate after failures, (2) small enough to fit in main memory, (3) updated frequently enough that it should not be stored on disk, and (4) unlikely to propagate the error that caused the system to fail.

This paper describes the motivation for the recovery box, its implementation, and some preliminary performance results. The first section gives statistics on the types of failures that occur in operating systems. It is these statistics on system failure types that lead us to believe the recovery box memory will be undamaged after the majority of software failures. If the recovery box memory has not been corrupted, then the system need not resort to a hard reboot. The section on implementation shows how Sprite and POSTGRES use the recovery box and explains the implementation and its memory layout. Finally, an evaluation section presents measurements of the recovery box's impact on recovery speed and regular execution performance.

Failure Statistics

New statistics on the frequencies of different types of system outages indicate that the recovery box will be intact and able to provide fast recovery from most failures. These statistics suggest that most failures are due to software errors, and that the most common types of software errors will leave the recovery box data undamaged.

Published data about the frequency of different kinds of outages is scarce, but a study of Tandem systems shows that faulty software is responsible for most failures [9]. Over time, Tandem systems have experienced fewer outages caused by hardware failures, environment failures, and operator errors. Software failures, on the other hand, have remained constant. Table 1 shows the percentages of each source of outage. In 1990, software errors accounted for 62% of Tandem system failures, while only 7% were caused by hardware. The trend towards faster increases in hardware reliability than in software reliability holds true in other environments as well.

Outage Sources	Percent
Software failures	62
Operator errors	15
Hardware failures	7
Environment failures	6
Scheduled maintenance	5
Unknown	5
Total	100

Table 1: Distribution of outage types. The table shows the distribution of types of outages occurring in Tandem systems between 1985-1990. Environment failures are caused by floods, fires, and long power outages.

Two studies that categorize the types of operating system software errors indicate that most such errors will not corrupt the recovery box [21, 22]. These two studies, summarized in Table 2, focus on the frequency of addressing errors, which are the errors that cause programs to corrupt memory. The BSD UNIX study divides errors into synchronization (47%), exception-handling (12%), addressing (12%), and miscellaneous (29%) errors. Exception-handling errors are those that occur in code for handling other errors, including transient hardware errors. The MVS study classifies errors in terms of low-level programming errors, of which 41% were "control" problems, 30% were addressing errors, 21% were miscellaneous, and 8% were data miscalculations. Control errors include such problems as deadlock, in which the program stops without corrupting anything but transient state. Data miscalculations include errors in which the wrong variable is used or a function returns the wrong value. Most of the errors classified as miscellaneous are related to performance or denial of service. Addressing problems do not cause the majority of software errors in either UNIX or MVS.

In addition, the MVS study shows that most memory corruption due to addressing errors is local to the data structure being manipulated. As shown in Table 3, at least 57% of addressing errors either corrupt the data structure the operating system intends to modify, or else corrupt memory immediately following the data structure. Only 19% of the MVS addressing errors covered in the study damaged parts of the system unrelated to the one where the error occurred. For example, a common type of addressing error in MVS is *copy overrun* in which a copy transfers too many bytes from one buffer into another, overwriting the data structure that follows the intended destination. A second common addressing error, called an *allocation management* error, occurs when the operating system continues to use a structure after deallocating it.

Error Classes	BSD UNIX	MVS
Addressing-related errors	12	30
Control-related errors	NA	41
Data miscalculation errors	NA	8
Synchronization-related errors	47	NA
Exception-related errors	12	NA
Miscellaneous errors	29	21
Total	100	100

Table 2: Software error type distributions. The table shows the distribution of software errors analyzed in studies of error reports from the IBM MVS and 4.1/4.2 BSD UNIX operating systems. The results columns give the percentage of errors that fall into each category. The two studies used different classification schemes, but both list addressing errors – the errors most likely to corrupt recovery box memory. The other error classes are described in the text.

With these statistics in mind, we have designed an interface to the recovery box that reduces the possibility of corruption from software failures. For example, corruption is unlikely if we use the recovery box to store back up copies of critical data structures, rather than allow clients to access recovery box memory directly. Also, an interface that does strict length checking when items are copied into recovery box memory will prevent copy overruns.

Unfortunately, the failure statistics do not give us enough information that we can protect against complex faults involving *error propagation*. If the system fails due to a logical error in one of its data structures, then storing this data in the recovery box could cause us to suffer the same failure after recovery. The only way we can protect the system from it is to choose data structures for insertion in the recovery box that have proven themselves over time to be relatively error-free. In commercial systems using a recovery box (and eventually in Sprite), data gathered from error reports would help indicate which data structures are prone to these propagated errors.

Implementation

To test the effectiveness of the recovery box, we have implemented it in Sprite and have used it for failure recovery of a Sprite file server and a POSTGRES database manager. POSTGRES runs as an application program on the file server and responds to requests from client programs running on other Sprite machines. Both Sprite and POSTGRES have features that allow fast recovery of disk data, so the recovery box experiments focus on recovery of distributed state. We first describe how Sprite and POSTGRES make use of the recovery box. Then we describe the interface through which the kernel and applications access the recovery box. Finally, we show the organization of the recovery box in memory.

Location of Damaged Area	Percent of MVS Errors
Near intended data	57
Anywhere in storage	19
Not evident from error report	24
Total	100

Table 3: Data corrupted by addressing errors. The table shows the relationship between the location of data corrupted by an addressing error in MVS and the location of the intended modification. Usually data corruption occurs near the data owned by the faulty code. In 24% of MVS error reports studied, this information was not evident.

How Sprite Uses the Recovery Box

Although the Sprite distributed operating system is UNIX-compatible at the system call interface, Sprite differs in some important ways from an NFS-style distributed UNIX system [17]. Unlike NFS file servers, Sprite file servers are not stateless. To achieve higher file system performance than NFS, Sprite caches file data on both client workstations and file servers. To maintain cache consistency, Sprite file servers keep information (in *handles*) for files that are open or cached on each client. In addition, the server keeps handles for objects other than files that are accessed through the Sprite file system, such as remote devices and pseudo-devices [24]. After a file server failure, this information must be regenerated before clients can continue opening and closing files. When a file server without a recovery box reboots, clients send the server a description of each of the handles for files and other objects that they have open or cached. The server regenerates its state information from this data sent to it by clients.

Handling this recovery information from the clients can overwhelm the server's processing capabilities, resulting in a *recovery storm* [4]. For a single Sun 4/280 file server with 40 clients, it currently takes over two minutes for all the clients to recover their distributed state.

Using the new recovery box eliminates this recovery-related client/server communication. During normal operation the server maintains structures containing data about each file or other object currently open or cached on each client. On average, the server maintains 10,000 to 15,000 such handles, or about 300 handles per client. From each of these structures the server preserves 52 bytes of essential information in the recovery box, making the server's recovery box space requirements less than a megabyte. Recovery box items must be updated every time a client opens or closes a file. In Sprite, all opens and closes are processed on the file server anyway, in order to maintain cache consistency, so finding the right place to call the recovery box functions was straightforward. We do not expect increased problems with error propagation, because the data stored in the recovery box for each handle is a subset of data that the server gathers from its clients during a hard reboot. When the server reboots using the recovery box, it rebuilds its tables of cache and file information with the data it retrieves from the recovery box. Without the recovery box, these tables would be regenerated by communicating with the client workstations.

An alternative to storing recovery information in main memory is to store it on the file server's disk. For instance, changes to the file handle information could be written to disk. However, it would be necessary to make these updates synchronously, since the file handle changes affect Sprite's distributed cache consistency protocol. Since this information changes on every file open or close, all open and close operations would incur the increased latency of a disk write. Furthermore, measurements of Sprite file system activity have shown bursts of file open and close requests as high as 100 per second – a rate too high to handle using disk storage without a group commit mechanism. Although a group commit would amortize the cost of the disk write across several opens and closes, it would still greatly increase the latency of some of the opens and closes. Another approach, used in Spritely-NFS [14], is to store information about active clients on the server's disk. After a failure, the server only needs to contact these active clients to regenerate the distributed file system state. The state of active clients changes slowly enough that there is no problem maintaining this information on disk, however, the recovery communication with clients could still be significant. The recovery box allows fast recovery without any conversations between client and server, without the complexity of a group commit, and without generating disk traffic on opens and closes.

The recovery box requires the addition of a small amount of new bookkeeping code in the Sprite file system. This code understands the contents of file handles and copies the relevant portions into items to store in the recovery box. It currently uses a hash table to map between file handles and their items. Since files can be open more than once simultaneously on the same client machine, the file system code also maintains reference counts in the hash table entries and in the recovery box items corresponding to file handles. This extra bookkeeping could be largely eliminated if we had the freedom to restructure Sprite's file handles to include the reference count and the ID that refers to the recovery box item.

How POSTGRES Uses the Recovery Box

Recovery performance in the POSTGRES database management system is dominated by the cost of reinitializing the DBMS server's connections with clients. In a conventional database management system, recovery includes the cost of write-ahead log processing (recovering disk state) in addition to client connection reestablishment. POSTGRES has an unconventional storage manager that maintains consistency of data on disk without requiring write-ahead log processing [20], so it does not need the recovery box to avoid this costly recovery step.

Without the recovery box, connection reestablishment is driven by the clients. When the database manager fails, all transactions executing on behalf of clients are aborted and all connection state held at the server is lost. Connection state includes, e.g., authentication information (secret keys or authentication tokens), client addresses, packet sequence numbers, and any packets queued at the DBMS.

When the server recovers, it must wait for the clients to detect that it has failed. After a connection times out, the client must reopen a connection with the DBMS and reauthenticate itself. POSTGRES can establish and authenticate (application-level) connections with only three messages, because it uses a sequenced packet protocol built on top of a datagram protocol (UDP), rather than a protocol based on streams (TCP/IP). After establishing the connection, each client must query the database to find out if its last transaction committed. Then, it must resubmit the transaction or take some other recovery action.

In a system with a recovery box, the DBMS stores authentication information and the client address associated with each connection in a recovery box item. The DBMS also stores the transaction ID of the last transaction it was executing on behalf of each client. Every time the DBMS begins a new transaction, it updates the recovery box item with the new transaction ID. Storing this transaction

ID on disk would be a bad idea, since POSTGRES is already disk bound for workloads with high transaction rates.

After a failure, the DBMS reinitializes its connection data structures from the backup data stored in the recovery box. Once the connections are reinitialized, the DBMS sends a message to each client indicating that (a) recovery has occurred (a new DBMS server ID is sent), and (b) a new sequence number has been chosen by the DBMS. The restart message also indicates the status of the last transaction that the DBMS executed on behalf of the client. If the message initiating the transaction was lost in the failure or if the transaction was aborted, the client must either resubmit the transaction or take some other recovery action. Authentication of the client is reverified when the DBMS receives the next message from the client.

POSTGRES does not use the recovery box to store any of the state associated with its storage system. Storage system performance optimizations requiring non-volatile RAM are discussed in [20]; for example, to reduce commit latency, committed data can be stored in non-volatile RAM instead of on disk. But this technique requires the operating system to guarantee that data stored in non-volatile memory be permanent. The recovery box does not make this guarantee, because if the system detects any errors during the fast recovery path, it will revert to the traditional disk-based recovery path and will discard the contents of the recovery box.

Interface

The recovery box interface is designed to help Sprite and its application programs manage backup data without exposure to some of the common software errors that corrupt main memory. For this reason we chose a structured and relatively inflexible interface; clients of the recovery box must explicitly insert, delete, and update recovery box items. In the structured interface, each item belongs to a type, and all items of the same type have common characteristics, such as size and checksum calculation routine. When a client creates a new type or a new item, the recovery box manager generates a unique ID (*typeID* and *itemID*, respectively). An itemID consists of the typeID and an *itemNumber*. Clients refer to types and items using these IDs.

We chose a structured interface over a more flexible one in which clients directly allocate and manage data structures in a reserved area of memory, because the structured interface provides more opportunity to avoid and detect recovery box corruption. Maintaining item size in the recovery box helps prevent the copy overruns that often caused storage corruption in MVS. Also, the recovery box can detect allocation management errors, because clients explicitly delete items when done with them. The structured interface also makes it easy for the recovery box to provide atomicity of operations. Atomic updates ensure that a system failure that interrupts an update will not cause checksum failure (and a hard reboot).

When an application program begins fast recovery, it must be able to find its recovery box items and begin regenerating state from them. To find these items, the application must be able to remember, across reboots, the type and item IDs assigned to its items by the recovery box. So that the application does not have to store the IDs on disk, the recovery box allows the application to choose its own well-known IDs and map from them to the system-assigned IDs. The application specifies its IDs when it initializes a type or inserts a new item. On recovery, the program maps from its application IDs to the system IDs once, at initialization, and not every time it accesses recovery box items. Allowing applications to specify their own IDs also facilitates sharing of recovery box items between cooperating UNIX processes.

The Sprite kernel and its applications use the same interface to the recovery box, except that applications call the recovery box routines via a system call. Table 4 lists the available functions. In order to initialize a new type of item, the system or application must call `InitItemType`. As parameters to this function, the caller specifies the maximum number of items that can be valid simultaneously, the item size, an optional applicationItemID, and a flag that signals whether checksums should be calculated and stored for items of this type. Applications can free an item type with the `DeleteItemType` function, but we do not expect applications to delete types often, except perhaps when a new application is being tested.

If creating and deleting types occurs frequently, these operations will have poor performance. Freeing a type from the middle of other allocated types can result in a fragmentation problem. The space consumed by the freed type may not be enough for any type allocated subsequently. If this occurs, all the item information and storage arrays can be shifted down (copied) in memory to leave enough space for the new type. The addresses in the per-type information must be updated to point to the new location of the item arrays. This shift operation does not affect clients of the recovery box, because the clients have no direct pointers to internal recovery box data. Shifting the item data may be costly, but it will only occur on type initialization. Another problem is that the shift is not an atomic operation. It is necessary to put a code in the current operation field of the header to signal whether a crash occurred in the middle of a shift. If so, the recovery box is unusable and the system will resort to a hard reboot. If clients of the recovery box need to delete types frequently, a different recovery box design will be necessary.

Operations on types	InitType DeleteType
Operations on single items	InsertItem DeleteItem UpdateItem ReturnItem
Operations on multiple items	InsertItemArray DeleteItemArray UpdateItemArray ReturnItemArray
ID mapping operations	GetTypeIDMapping GetItemIDMapping

Table 4: Recovery box operations. This table lists the operations available through the recovery box interface. The first set of functions applies to item types; the second set applies to individual items, and the third to multiple items. The last set of functions returns the recovery box typeID or itemID when given an application's typeID or itemID.

Interface functions for operating on individual recovery box items include: `InsertItem`, `DeleteItem`, `UpdateItem`, and `ReturnItem`. To insert an item in the recovery box, the caller provides the item's typeID, a pointer to the data for the item, and an optional application itemNumber. DeleteItem frees the space in the recovery box allocated for an item. To update the contents of an item in the recovery box, the caller must provide the itemID and a pointer to the new data for the item. ReturnItem returns a copy of the specified item in a buffer provided by the caller.

Additional interface functions, such as `InsertItemArray` and `ReturnItemArray`, allow applications to operate on multiple items, avoiding the system call overhead that would be incurred by multiple operations on individual items. To insert multiple items the caller must provide the typeID, the number of items it wishes to insert, an optional array giving the application's itemIDs for each item, and an array of items to insert. The function returns an array giving the new system-assigned itemIDs for the inserted items. ReturnItemArray returns copies of all the items for an item type and an array of itemIDs. If there is insufficient space in the buffers given to it as parameters, the function returns an error and the amount of room required for each of the buffers.

Recovery Box Structure

The recovery box is organized in memory to provide fast insert and delete operations and fast access to items and their type information. Figure 1 shows the layout in memory. There are three sections of the recovery box: a header, followed by an array of per-type information, followed by the item area. In the item area there are two arrays for each item type: an array of per-item information and an array storing the items themselves. The type information, item information and item storage are all implemented as arrays for fast access given an array index; the recovery box typeIDs and itemNumbers are actually indices into the per-type and per-item information arrays, respectively. As described below, a portion of the per-item information array implements a free list of unallocated items for each type, providing constant-time item insertion and deletion.

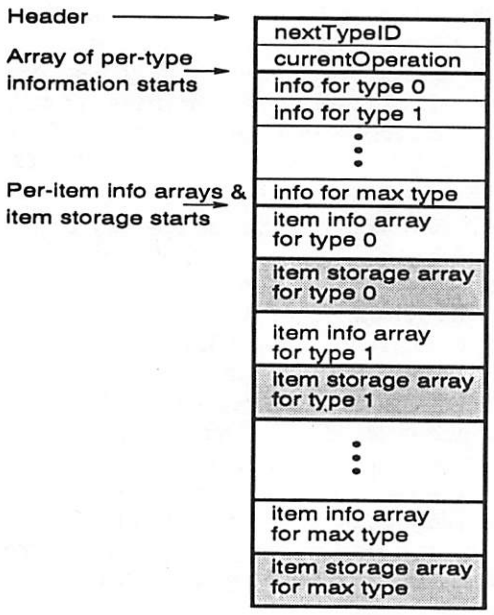

Header
Array of per-type information starts
Per-item info arrays & item storage starts
nextTypeID
currentOperation
info for type 0
info for type 1
info for max type
item info array for type 0
item storage array for type 0
item info array for type 1
item storage array for type 1
item info array for max type
item storage array for max type

Figure 1: Layout of recovery box in memory. The recovery box layout in memory starts with a header that gives the next typeID to allocate, and provides a code for the current operation. The current operation code is used to ensure atomicity of recovery box insertions, deletions, and updates. The header is followed by an array that gives information about each type of item stored in the table. The per-type information is followed by the per-item information array for the first item type. Following the per-item information for each item type is the array of the items themselves, shown as a shaded entry in the figure.

The header at the very beginning of the of the recovery box contains information that must be saved across fast reboots. The first field in the header, *nextTypeID*, gives the index of the next typeID to be allocated. The second field in the header specifies the current operation on the recovery box in order to ensure that insert, delete, and update item operations are atomic. (Other functions, except those operating on multiple items, are already assured of being atomic.) At the beginning of an insert, delete or update operation, this field is set with a code for the current operation. In

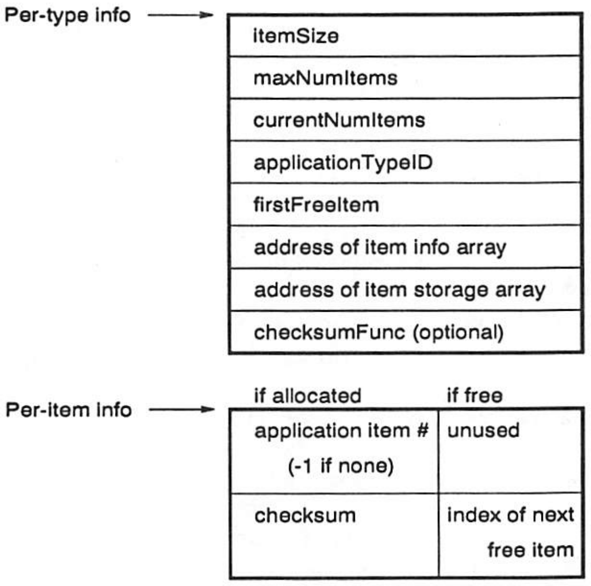


Figure 2: Contents of type and item information arrays. The top half of the figure shows the contents of an entry in the per-type array. This array lists, for each type, the size of the items, the maximum number of items that can be allocated, the current number of allocated items, the itemNumber of the first free item, the memory address for the per-item information array, the memory address for the item storage array, and a possible checksum routine. The lower half of the figure shows the contents of an entry in the per-item information array. This array lists, for each item, the application itemNumber, if one exists, and a possible checksum value for the item. If the item is not allocated, the checksum field instead gives the itemNumber of the next unallocated item.

the case of a delete or update operation, it also includes the target itemID. The field is not cleared until the operation completes, making it possible to detect and back out of incomplete operations. At the beginning of an update, the original value of the item is copied to an extra item space at the end of the item storage array. If the machine crashes during the update, the original value of the item can be retrieved.

The recovery box header is followed by an array, accessed by typeID, that gives information about each item type. The top half of figure 2 shows the information stored in an entry in this array. Each entry lists the item size, the maximum possible number of items, the current number of items, the application typeID, if any, the index in the item storage array of the first free item, the memory address of the per-item information array, the memory address of the item storage array, and a pointer to a checksum routine. If a type's checksum routine field is zero, then no checksums are calculated for that type's items. The generic checksum routine must take as a parameter the size in bytes of the item to be checksummed. However, the checksum routine for the type containing the Sprite kernel's file handle items uses a checksum routine unrolled to optimize checksumming the 52-byte file handle items.

The per-item information arrays are used for fast item allocation and for storing checksums and application itemNumbers. The lower half of figure 2 shows the contents of an entry in such an array. Each entry in the array consists two fields. The first is only valid if the item has been allocated. It contains the application itemNumber, if the application provided one while inserting the item, or a −1 if there is no application itemNumber. The second field has a different meaning depending on whether the item has been allocated or not. If the item has been allocated with a checksum performed on it, the field contains the checksum result. If the item has not been allocated, the field contains the index of the next unallocated item in the array, thus implementing a free list of items. For the last item on the free list, this field is −1. The index of the first free item in the list is stored in the per-type information. Upon deleting an item, its itemNumber is added to the beginning of the list. The free list makes finding free spaces in which to insert new items fast.

Storing the application itemNumbers in the per-item array provides quick mapping from the recovery box system's itemIDs to the application's itemIDs; however, mapping from application IDs to system IDs is not particularly fast. This is why none of the functions in the recovery box interface take the application itemID, except for the function that returns the system itemID given the application itemID. Applications can maintain their own tables to do this mapping quickly, but we leave this functionality outside of the recovery box for the sake of simplicity.

Besides considering the internal memory structure of the recovery box, we must also be able to find the recovery box's memory location after a fast reboot. Our recovery box is always allocated at the same virtual address, so it is easy to locate. Also, the addresses inside the recovery box (specifying the per-item information arrays and storage areas) remain valid without modification across fast reboots. The virtual address of the recovery box must also map to the same physical address upon every fast reboot so that it points to the physical memory containing the recovery box data. The operating system must avoid clearing and initializing this area of memory on a fast reboot. The memory for the recovery box could be battery-backed RAM, but in the absence of non-volatile memory it can be any reserved area of system memory.

We chose to locate the recovery box in a special area of the kernel text segment in order to minimize damage due to memory corruption. The memory pages for the recovery box are marked writable as well as readable, in contrast to the rest of the text segment. We chose the text segment because it reduces the sources of possible system address errors that could overwrite the recovery box. Except for the recovery box, there are no writable data structures in the text segment and no sources of pointer manipulation using text segment addresses. Since error statistics show that most addressing errors are localized around the intended data structures, this should eliminate most addressing errors except for any caused by manipulation of the recovery box itself.

The three main shortcomings in our current recovery box implementation are its lack of access protection, its static memory allocation, and its lack of atomicity for operations on multiple items. At present, there is no security provided by the recovery box, except that only applications with root privilege can access items allocated by the kernel. The static memory allocation imposes a limit on the number of item types that can be initialized and a limit on the overall size of the recovery box. This is why the maximum number of items that will be valid simultaneously for a type must be specified when that type is initialized. The type initialization function returns an error if there is insufficient space for the desired number of items. Currently, the size of the recovery box is compiled into the operating system. If the recovery box resides in non-volatile memory, such as battery-backed RAM, it is likely that the amount of non-volatile memory will already impose a size restriction. Even without such a restriction, placing the recovery box in a static area of system memory, such as the text segment, makes it difficult to expand the recovery box in physical and virtual memory while guaranteeing the same virtual/physical address mapping across reboots. Finally, operations on multiple items are currently not atomic. It would be easy to change our implementation to provide atomicity of multiple item inserts, but providing atomicity of multiple updates and deletes would significantly increase the complexity. While neither the operating system nor the applications we considered would benefit from atomicity of multiple updates, this could be a worthwhile problem to tackle in other environments.

Evaluation

In this section we evaluate the recovery box implementation based on its effect on reboot times and on regular execution performance. To improve reboot times, we use the recovery box to rebuild the operating system and DBMS distributed state, but we have also used a variety of techniques to improve other steps in the reboot sequence. With the recovery box, combined recovery time of Sprite and POSTGRES is about 32 seconds. If the recovery box has been corrupted by a software error or power failure, recovery time will still take many minutes. The effect of the recovery box on regular execution is not large, about 5% overhead on Sprite open/close file operations and less than 1% on POSTGRES debit/credit transactions. The Sprite overhead could be reduced to 2% with optimizations that are not possible in our current development environment. Unless otherwise specified, all measurements were done on a SPARCstation 2 file server (40 Megahertz, 20 integer SPECmarks).

Sprite/POSTGRES Recovery Speed

Although the most lengthy step in rebooting Sprite is distributed recovery, to which we can apply the recovery box, the server must also load the system text and initialized heap, initialize the kernel modules, check its disks, and initialize the daemon processes. To provide fast reboot and recovery, we have reduced the time consumed by all of these steps. Below, we list the steps in the reboot and recovery sequence, along with our improvements and the reduction in time consumed by each step.

(1) Retrieve operating system text and data – The first step in the reboot sequence ensures that an undamaged copy of the operating system text and initialized data is in system memory. Reading the Sprite operating system text and initialized heap from disk into memory requires approximately 20 seconds on a Sun 4/280. This is because the program that loads the kernel image from the disk only manages to read about 50 kilobytes per second. Other systems' disk boot programs might be more efficient. We reduce this cost to much less than 1 second, on both the Sun 4/280 and the SPARCstation 2, by using a technique similar to the recovery box. The operating system text lies in a write-protected area of memory, namely the text segment. Upon fast reboot, we simply reuse the text segment already in memory. Because the initialized data (heap segment) is not write-protected, we must do some extra work to preserve a non-writable copy of it. The hard reboot startup code makes a copy of the kernel initialized data in an area of the text segment that it then write-protects. The fast reboot startup code then copies this initialized heap data from its write-protected storage to the correct address in its heap segment. The system initiates a fast reboot sequence by jumping to a special text address and starting execution at that point rather than downloading a new kernel image and starting execution at the very beginning of the text segment. Because the text segment and copy of the initialized heap data are write-protected, no checksum is performed on this data. Preserving the kernel text and initialized heap data in memory also speeds recovery of machines that ordinarily download the kernel image over the network.

(2) Initialize kernel modules – In addition to the initialized heap data stored in a special area of the text segment, we also store information that would otherwise need to be recomputed when kernel modules are initialized, for example, the machine's internet address. Storing these items shortens our module initialization from about 7 seconds to less than 2.

(3) Check disks – We have converted our disks over to use the Log-Structured File System (LFS) [16]. Without LFS, a hard reboot of a file server with 5 gigabytes of storage using a traditional UNIX file system such as the 4.2 BSD Fast File System [13] can take up to 40 minutes to restore the consistency of file system metadata. LFS does not require a file system check (`fsck`) during system initialization, because it leaves its disk-resident file system metadata consistent even after a crash. LFS recovers from failures by rolling forward from a log on disk which is checkpointed at least every 30 seconds. The cost of rolling forward depends on the number and type of I/O operations written to the log since the last checkpoint; two seconds is a conservative estimate of LFS recovery time given current workloads. While LFS does not need a file system check to make its metadata consistent after a failure, it could use one to check that its directory structure has not been damaged by errors. A version of LFS to be included in a future BSD UNIX release [18] uses `fsck` to check the consistency of essential directories on reboot, such as those on the root file system.

(4) Recover distributed state – The Sprite file servers must regenerate the distributed file system state they were using prior to a crash. For 40 clients accessing a single Sun 4/280 file server, this recovery takes more than two minutes depending on how heavily-used the system is at the time of the crash. Using the recovery box on a SPARCstation 2 file server with 20,000 pieces of distributed state, this portion of the recovery time is ten seconds. We chose 20,000 pieces of distributed state for our tests, because this amount is greater, by one-third to one-half, than the number of file handles needed by Sprite's main file server with 40 clients. The ten-second recovery time includes the cost of some disk accesses to retrieve descriptor information such as file permissions and last access times. If we added the necessary descriptor information to the recovery box file handle items, or if we recovered the information in a lazy fashion as needed, then we could eliminate the disk accesses and reduce this portion of our recovery times further. A direct comparison of our measurements for distributed state recovery times with and without the recovery box is not fair, because the Sun 4/280 file server is a slower machine than the SPARCstation 2. We are currently unable to substitute a SPARCstation 2 for our Sun 4/280 file server. However, the recovery box reduces distributed state recovery by at least an order of magnitude, even on the SPARCstation 2.

(5) Kernel and daemon processes – The start-up of internal kernel processes and various system daemons, such as sendmail, inetd and lpd, currently requires about 40 seconds. Although we could potentially store some of the state of necessary processes in the recovery box, we currently just start up the most crucial processes (such as POSTGRES and login) first, consider the system to be "up" at that point, and then start up the other daemons in a lazy fashion. This has reduced the process start-up time for necessary processes to less than 10 seconds.

The total time required for a fast reboot after our improvements is about 26 seconds on a SPARCstation 2. This includes 10 seconds for distributed state recovery, 2 seconds for disk initialization and 14 seconds for the other steps. This compares with the several minutes required for a hard reboot.

The most lengthy step for POSTGRES recovery is reinitializing the server's connections with client applications. Without the recovery box, POSTGRES clients discover failures using timeouts. After the timeout, each client must query the database to find out whether its last transaction committed. The timeout alone requires several seconds since it must allow for worst-case queueing delays before assuming that the DBMS is down.

To measure POSTGRES recovery times, we ran a debit/credit benchmark based on TP1 [2], but to expedite the measurements we used a much smaller database than TP1 requires. A single POSTGRES DBMS managed the database from a Sprite file server. Ten POSTGRES client processes running on a single Sprite client machine generated the transactions. For our experiments, we used a version of the DBMS that was single-threaded. While the DBMS executes a transaction for one client, transactions sent by the others are queued in the DBMS address space. Single-threaded execution means that adding POSTGRES clients increases recovery time (due to client connections) without increasing the overall transaction rate of the DBMS.

Breaking down DBMS recovery into its component parts, we get the following recovery times:

(1) Demand page DBMS code and load system catalogs from disk – We have made no optimizations here so far, although both the DBMS text segment and some parts of the catalogs could be cached in the recovery box. The cost of this initialization is about one second.

(2) Initialize internal memory data structures – POSTGRES must initialize many hash tables and linked lists. Together, all of this initialization only takes 0.4 seconds, so we have made no optimizations here.

(3) Restore consistency of database – Using the POSTGRES storage system, POSTGRES ensures that disk data is always consistent; updates caused by committed transactions are reflected in the database and updates caused by uncommitted transactions are not. As a result, it does not have a step analogous to Sprite's disk check. A conventional DBMS does have such a step and can spend many minutes restoring the consistency of its disk data from a write-ahead log, depending on the length of time between checkpoints and the DBMS transaction rate.

(4) Recover client connections – With a fast reconnect protocol that relies on the recovery box, POSTGRES clients are notified immediately of a DBMS failure and can resubmit lost transactions with a single message exchange. The time to recover 10 client connections using this protocol is less than a second. Other systems have implemented client recovery with add-hoc mechanisms involving several message exchanges, queries of the database, and sometimes human intervention (including the original POSTGRES implementation). The times required by these mechanisms vary widely but all will be longer than a single message exchange per client.

Using the recovery box, total recovery time for POSTGRES and 10 clients (ignoring operating system recovery time) is about six seconds. Most of this cost comes from reloading the database into main memory.

Regular Execution Performance

Using the recovery box has not significantly reduced the regular execution performance of Sprite or POSTGRES. Table 5 shows the breakdown of time required for the Sprite file system recovery box operations. The table gives measurements for the recovery box code itself, for the file system layer that calls the recovery box code, and for the file open and close times seen by a SPARCstation 2 client of the file server, with and without the recovery box. The file open/close measurements include the time for the kernel-to-kernel remote procedure calls between the client and file server. We report measurements in terms of pairs of operations – recovery box item insert/delete operations and file open/close operations.

For a SPARCstation 2 Sprite file server, the time to insert, checksum and delete a file handle item in the recovery box is 28 microseconds on average. The checksum calculations require 4 microseconds per file handle. The open/close test performs two insert operations during file open and performs two delete operations during file close for reasons explained below.

The file system bookkeeping code that calls the recovery box requires more time for an insert/delete operation: 72.3 microseconds on average. In part, this is due to the time to set up the items to insert in the recovery box, but it is also due to a problem in our current implementation. Inserts and deletes of file handle items currently require some extra bookkeeping and a hash table lookup. The hash table maps from file handles to itemNumbers and recovery box item reference counts. Much of this extra code would be unnecessary if we could avoid the mapping by changing the format of the file handle structure in Sprite to include the itemNumber and recovery box reference count. Unfortunately this modification would require recompiling and rebooting the entire Sprite cluster. The resulting outage would affect dozens of irritable graduate students and several aggressive faculty members. We have postponed making these changes until the recovery box prototype has proven itself to be stable.

Operation	Avg time in microseconds	Standard deviation
Recovery box insert/delete with checksum	28	± 0.0
Recovery box insert/delete with no checksum	24	± 0.0
FS insert/delete with checksum	73	± 1.9
FS insert/delete with no checksum	72	± 2.3
Open/close with with recovery box	3616	± 55.0
Open/close without recovery box	3450	± 86.6

Table 5: Sprite recovery box performance. The first two entries give the time to insert and delete a file handle item in the recovery box, with and without a checksum. The second two entries give the time to insert and delete a file handle, including the extra overhead in the file system ("FS") bookkeeping code. The last two entries give the time to execute a file open/close operation from a client workstation, with and without a recovery box running on the file server. This time includes the kernel-to-kernel remote procedure calls between the client and file server. The results columns give the average and standard deviation of 10 sets of measurements. Each measurement executed 100 iterations of 50 open/close or 50 insert/delete operation pairs. The recovery box already contained 1000 file handle items before the measurements started.

The latency experienced by a client opening and closing a file on a server with a recovery box includes twice the cost of the file system bookkeeping code, because each file open/close pair requires inserting and deleting two file handle structures in the recovery box. One file handle contains data about the file on disk; this is similar to a UNIX inode but also includes distributed cache consistency information. The other handle is a reference to the open file and contains information similar to a UNIX open file table entry. This second handle is called a *stream handle*. The ability of processes in Sprite to migrate to idle machines [8] means that two processes on different machines may share the same reference to an open file. The server maintains the shared file offset in the stream handle and must be able to regenerate this information after a reboot.

Operation	Average time in microseconds	Standard deviation
Recovery box update	55	± 2.3
System call overhead	19	± 0.3
Checksum 92 bytes	10	± 0.0
Copy 92 bytes	8	± 0.1

Table 6: POSTGRES recovery box performance. The first entry shows the time to update a POSTGRES client connection in the recovery box. The last three entries give the major components of this cost – system call overhead, checksum calculation, and copying costs. The results columns give the average and standard deviation of 10 sets of measurements. Each measurement is the average cost of 100,000 operations.

While the recovery box adds a 5% latency to the open/close times seen by a client workstation, the effect on file server throughput is not as large. On average, the main Sprite file server for 40 clients receives two file open/close pairs per second. The current recovery box implementation thus adds, on average, an extra 332 microseconds of server processing per second, or less than a 0.1% increase in processing demand at the server. However, file system activity is bursty, and the server sometimes sees as many as 50 file open/close pairs in a second. During peak activity, the recovery box would thus add 8.3 milliseconds of server processing per second, for a potential 0.8% reduction in server throughput. If we were able to eliminate much of the overhead in the file system bookkeeping code on the file server, the overhead, including increased latency seen by the clients, could be cut in half. We believe this means the recovery box overhead would be acceptable, even in a high performance system.

POSTGRES updates a 92-byte item in the recovery box on each transaction; the entire connection structure is updated even though only the transaction ID changes. Table 6 shows the breakdown of times for POSTGRES recovery box operations. Individual POSTGRES recovery box operations are slower than the ones made by Sprite. POSTGRES must pay the overhead of system calls. Because it is inserting larger objects into the recovery box, the cost of copying data and computing checksums also increases. The operating system is able to use an optimized checksum routine for file handles (with an unrolled loop) while application programs must use a generalized one. POSTGRES could, of course, compute and check its own checksums if the difference in performance were critical. In fact, the measured cost of the recovery box operations is much smaller than the standard deviation between POSTGRES transaction execution times, so there is no reason to further optimize checksum calculation.

Availability and Reliability Impact

Our testbed has shown that the recovery box has acceptable performance, but its impact on system availability and reliability will determine the viability of the technique. We have just finished the implementation and initial test runs of the recovery box. We hope to have it in general use in the next couple of months. After that, collecting data on its effectiveness should take six to twelve months, given our current system failure rate of about once a week. If the recovery box provides fast recovery from our most common failures without reducing the reliability of the system, then we will consider it to be a successful tool.

Conclusion

We can provide high availability in the UNIX environment without great expense or performance degradation by using a recovery box to store state that would otherwise be regenerated from scratch by a system reboot. Failure statistics indicate that memory used for the recovery box is unlikely to suffer corruption due to system failure. The system can fall back to the traditional, slower, recovery path if it detects any problems with the backup recovery data during reboot. This reduces down-time while avoiding the complexity of more sophisticated fault-tolerant techniques. The recovery box concept can be applied to any system in which it is possible to identify and isolate the data structures that (1) are expensive to regenerate from scratch, (2) are small, (3) are updated too frequently to store on disk, and 4) do not have a history of causing system failures. As long as memory requirements do not become outrageous, the recovery box should scale well to large distributed systems, since it substitutes local memory accesses for extra network communication or disk accesses during restart.

Currently, we have only used the recovery box for Sprite and POSTGRES, but we believe that it would be useful in many applications with state that is expensive to regenerate. The internet name servers and programs with TCP connections are examples of applications with distributed state that could be stored in the recovery box. The recovery box might also be an inexpensive way to limit the number of editing changes lost in a vi session due to a system failure. Especially when implemented with non-volatile RAM, the recovery box could help eliminate file system consistency checks in non-LFS file systems. It should also be possible to apply it to user-level servers, such as those running on top of the Mach microkernel [1], and to other distributed application programs.

Acknowledgements

We would like to thank John Ousterhout, Mendel Rosenblum, Margo Seltzer, Jim Mott-Smith, Joe Hellerstein, Mike Olson, Jeff Mogul, and Adam Glass for their very helpful comments on drafts of this paper. Mark Weiser, Keith Bostic, John Wilkes, John Hartman and Wendell Baker all made useful suggestions about the recovery box design. Mike Kupfer quelled periodic panics. The work described here was supported in part by the National Science Foundation under grants CCR-8900029 and MIP-8715235, and the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

References

1. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M., Mach: A New Kernel Foundation for UNIX Development, *Proceedings of the Summer 1986 USENIX Conference*, June 1986.
2. anon, A Measure of Transaction Processing Power, in *Readings in Database Systems*, Morgan Kaufmann Publishers, 1988, 300-312.
3. Auslander, M., Larkin, D. and Scherr, A., Evolution of MVS, Vol. 25, September 1981.
4. Baker, M. and Ousterhout, J., Availability in the Sprite Distributed File System, *Operating Systems Review 25*, 2 (April 1991).
5. Bartlett, J., A NonStop Kernel, *Proceedings of the 8th Symposium on Operating System Principles*, December 1981.
6. Bhide, A., Elnozahy, E. and Morgan, S., Implicit Replication in a Network File Server, *IEEE Workshop on Management of Replicated Data*, November 1990.
7. Borg, A., Blau, W., Graetsch, W., Herrman, F. and Oberle, W., Fault Tolerance Under UNIX, *ACM Transactions on Computer Systems 7*, 1 (February 1989).
8. Douglis, F. and Ousterhout, J., Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software -- Practice & Experience 21*, 8 (July 1991).
9. Gray, J., A Census of Tandem System Availability between 1985 and 1990, *IEEE Transactions on Reliability 39*, 4 (October 1990).
10. Jewett, D., Integrity-S2 -- A Fault-tolerant UNIX Platform, Field Failures in Operating Systems, *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.
11. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L. and Williams, M., Replication in the Harp File System, *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.
12. Long, D., Carroll, J. and Park, C., A Study of the Reliability of Internet Sites, *Proceedings of the 10th Symposium on Reliable Distributed Systems*, September 1991.
13. Mckusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S., A Fast File System for UNIX, *ACM Transactions on Computer Systems 2*, 3 (August 1984).
14. Mogul, J. C., A Recovery Protocol for Spritely NFS, *To appear in the Proceedings of the USENIX Workshop on File Systems*, May 1992.
15. Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B., The Sprite Network Operating System, *IEEE Computer 21*, 2 (February 1988).
16. Rosenblum, M. and Ousterhout, J., The Design and Implementation of a Log-structured File System, *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.
17. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., Design and Implementation of the Sun Network File System, *Proceedings of the the Summer 1985 USENIX Technical Conference*, June 1985.
18. Seltzer, M. I., Personal Communication, April 1992.
19. Stonebraker, M. and Rowe, L., The Design of POSTGRES, *Proceedings of the 5th ACM SIGMOD Conference*, June 1986.
20. Stonebraker, M., The POSTGRES Storage System, *Proceedings of the 13th International Conference on Very Large Data Bases*, September 1987.
21. Sullivan, M., *Unpublished survey of software errors reported in 4.1 and 4.2BSD UNIX*, 1990.
22. Sullivan, M. and Chillarege, R., Software Defects and Their Impact on System Availability -- A Study of Field Failures in Operating Systems, *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.
23. Webber, S. and Beirne, J., The Stratus Architecture, *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.
24. Welch, B. and Ousterhout, J. K., Pseudo Devices: User-Level Extensions to the Sprite

File System, *Proceedings of the the Summer 1988 USENIX Technical Conference*, June 1988.

Author Information

Mary Baker is a Ph.D. student in computer science at the University of California at Berkeley. She currently works as a research assistant on the Sprite network operating system project directed by Prof. John Ousterhout. Her interests include operating systems, distributed systems, computer architecture, and raising the cholesterol levels of friends and colleagues. Her email address is mgbaker@cs.berkeley.edu.

Mark Sullivan will receive his Ph.D. in computer science from the University of California at Berkeley in Summer 1992. He currently works as a research assistant on the POSTGRES project under the direction of Prof. Mike Stonebraker. His interests include database management systems, fault tolerance, operating systems and travel to exotic countries. His email address is sullivan@cs.berkeley.edu.

Both authors can be reached via U. S. Mail at the Computer Science Division, 571 Evans Hall, U. C. Berkeley, Berkeley, CA 94720.

On Migrating a Distributed Application to a Multi-Threaded Environment

Thuan Q. Pham, Pankaj K. Garg – Hewlett-Packard Laboratories

ABSTRACT

Light-weight computation threads in a multi-threaded operating system promise to provide low-overhead computation and fully sharable addressing space not available in conventional process-oriented operating systems. Traditional distributed applications based on processes can be re-architectured to use concurrent threads in a multi-threaded platform to take advantage of faster context switches and shared-memory communication.

We investigated this expectation by porting an existing distributed application to a multi-threaded environment. As a result, we virtually eliminated the cost of message-based IPC, replacing it with shared-memory communication between threads.

In this paper we address the benefits, the difficulties, and the trade-offs of such a re-architecture. We also comment on some feasible architectures for migrating currently distributed applications to multi-threaded environments.

Introduction

The usual process abstraction [Sal66] has too many things anchored to it to meet the needs of aggressively concurrent applications. Process creation and context switching result in high overhead on the part of the operating system, often using far more resources than one would like [ABB+86]. Furthermore, since the processes do not share resources, distributed applications with significant data sharing must communicate via expensive message-based interprocess communication (IPC) mechanisms.

Earlier efforts have tried to circumvent these problems by utilizing coroutine and user thread packages to simulate and manage multiple contexts, with shared memory, within a single process [Mar90, JRG+87]. However, since the kernel has no knowledge of such coroutines or sub-processes, these coroutine packages cannot take advantage of the operating system's scheduling services, and an application using them cannot utilize more than one processor in a multi-processor environment. Thus the questions of alleviating expensive context switching and expensive interprocess communication are not completely addressed by user-created coroutines and sub-processes.

The above problems have been addressed by operating systems supporting light-weight threads and shared resources (e.g., OSF1). The creation and maintenance of threads require lower operating system overhead than processes. A thread, when created, has access to all the process information in the task[1]. Since the computation threads share all resources within a task, including the memory address space, ''inter-process'' communication can be done cheaply and efficiently with shared memory.

A systematic reduction of heavy-weight processes to light-weight threads with shared memory, whenever possible, provides a substantial improvement in performance [ABB+86, FR86, JRG+87, TR87]. However, the threads facility restricts the architecture of distributed systems. Performance improvements come at the expense of the lost generality to the process model: processes are machine independent; threads may not be. Further cost is incurred by the effort spent in porting existing software systems to a new, multi-threaded platform. These issues are investigated by our re-architecture of an existing distributed application from a single-threaded environment to a multi-threaded one. We report on the problems and benefits of such a re-architecture.

The Experiment

The distributed application used in this experiment is *Matisse*, a knowledge-based team programming environment derived from the Workshop System [Cle88]. *Matisse* offers automated support for communication and coordination in team programming. Its architecture is illustrated in Figure 1. The core of each unit of *Matisse* includes an expert system, an object editor, and a graphical object browser, all residing in the programmer's workstation. In the current implementation, all components of *Matisse* are separate processes that run concurrently and share information via sockets. By merging some of these UNIX processes into threads within the same task, we can obtain significant performance improvement with light-weight threads and shared-memory IPC.

[1]A task with one thread is equivalent to a process. We use *task* to denote a multi-threaded process.

For this experiment, we have ported and merged only the *Workshop* and *Editor* processes (Figure 2). These processes are prime candidates to receive the benefits of the merge because they naturally reside on the same machine and share a large collection of data. The locality of these processes enable them to be merged without any loss of generality or usefulness, and their interprocess communication can benefit from direct memory sharing.

In addition to providing performance measurements, the experiment gives us some understanding of the difficulty of this porting process, including the conditions and requirements that make the operation possible and optimal (data representation, locality, garbage collection, etc.). These rules of thumb might be a helpful guide in identifying a suitable architecture, or re-architecture, of processes and threads for distributed software systems in the future. Our experience suggests that performance improvements do not come without costs and it is up to the designer to judge whether this approach is feasible for specific applications.

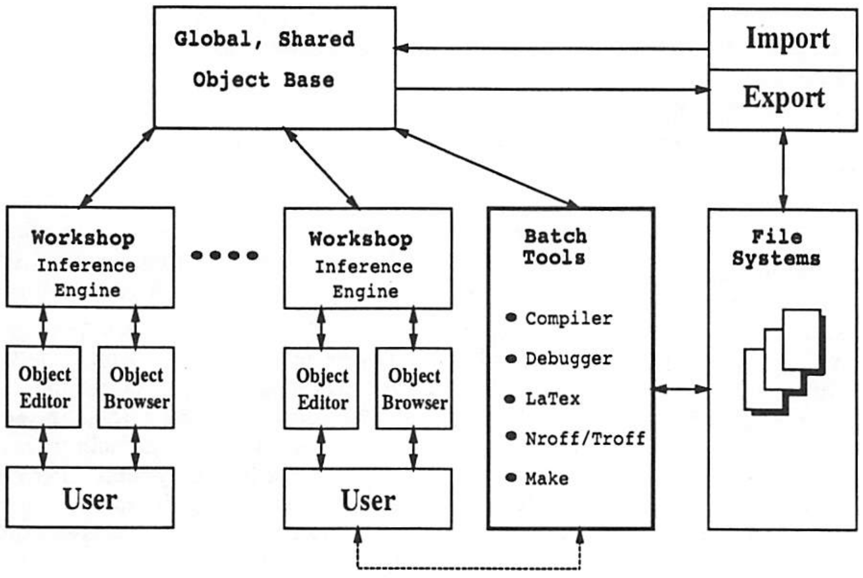


Figure 1: *Matisse* Architecture: A central shared information base is shared by all team members. Each person has an individual (active) information base which talks to various interactive tools.

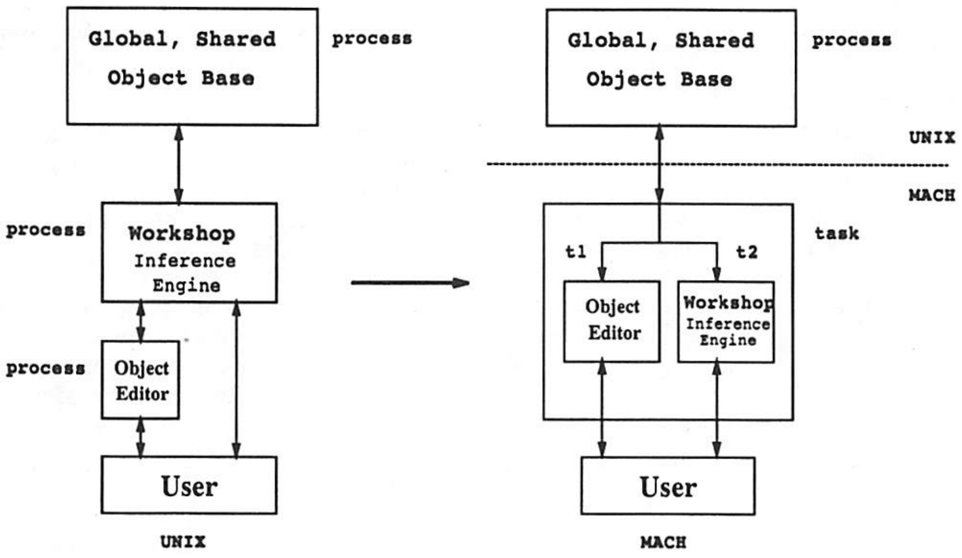


Figure 2: *Matisse:* experimental subset

System Re-Architecture

The experiment is divided into three stages. The first stage was a "straight port" of the Editor and the Workshop to the multi-threaded operating system, making each process a single-threaded task in the new environment. The second stage involved merging these two single-threaded tasks into one task with two threads while leaving their interprocess communication mechanisms undisturbed. Finally, in the third stage, we replaced most of the IPC messages between the two threads with some primitive routines that allow direct access to shared data in memory, and a new communication protocol using these primitives. Figure 3 depicts the stages of the experiment, showing the shared memory and illustrating also the process, task, and thread boundaries. The division of the experiment into these three stages is natural, since each stage has its own set of problems that, for the most part, is distinct and unrelated to the others. The following sections describe each phase of the experiment, discuss the difficulties that were encountered, and present our solutions.

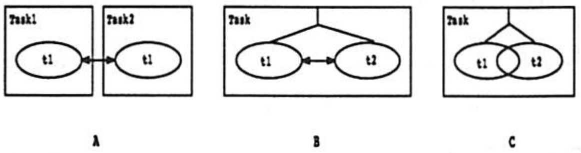


Figure 3: Re-architecture process: from disjoint single-thread tasks to multi-threads task with memory sharing capability; a) single-thread tasks, b) multi-thread task, c) multi-thread task with memory sharing.

From Processes To Single-Thread Tasks

In stage 1, we had the problem of having different names for [equivalent] system calls and a different file directory hierarchy. This can be rectified by systematically checking, identifying, and tracking down the right functions and files in the new environment. Although this was the simplest and most straight-forward step of the experiment, it was, however, not necessarily easy, depending on the portability of the existing system. This step could be as simple as a recompilation, or as arduous as a major rewrite. For example, our Workshop port went particularly smoothly, requiring little beyond setting up the directories, the Makefile, and the recompilation. On the other hand, our Editor, a modified Emacs [Sta84], was particularly hard to port since its complex code exploited many system-specific features, and exposed numerous system differences.

An interesting problem we encountered was the difference of implementation of certain operating system services. One example was the incompatible side-effects of the functions *regex/regcmp* across the two operating systems. One library routine stored the data to be processed in an internal static structure, and thus could not be called recursively, while the other took the argument from the program stack and contained no static internal state. It was not possible for us to re-implement such services to suit only our needs since other existing programs depended on them. Our solution was to define the functions and data structures necessary to use instead of the existing resources.

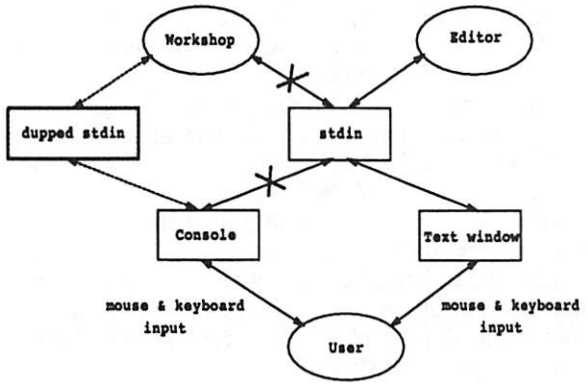


Figure 4: resource contention: Although the *Editor* has it own editing window and does not use the shell window that the other thread uses for input (which is really *stdin*), its code is written in such a way that the core of the *Editor* expects user inputs from *stdin* by mapping the channel associated with the editing window to *stdin*. The two threads' dependence on the input stream created a confusion that led to system failure. The communication channel *stdin* was "dup"ed and the copy was given to the Workshop process to avoid contention.

Merging Single-Thread Tasks To A Multi-Thread Task

In stage 2, we created a new *main* function to set up global, shared resources such as thread IDs, mutex and condition variables, and then fork the two execution threads. Since the interface allows only one argument to be passed to a thread at creation time, the *main* function must parse the command-line arguments, package them in data structures, and hand them to the appropriate thread creation routine.

I/O Contention

A thread-unsafe situation resulted when the merged threads both wanted to access the same file descriptor (most commonly *stdin*). To give one thread exclusive control of the file descriptor would require a major rewrite of the other thread. Our solution was to *dup* the overlapping file descriptors and give each thread a different handle. As a result, the two threads no longer contend for the same file descriptor. This problem is depicted in Figure 4.

Problems With UNEXEC

Applications often use a mechanism called *unexec* to freeze the image of the process and dump it out to an *a.out* format file capable of being restarted. Making *unexec* work for a multi-threaded task requires that we properly start up the threads, coordinate their terminations, and clean up after they exit.

First, for the program to be restarted after an *unexec*, we must call the thread initialization routine to set up the internal system resources needed for operation because the individual thread states are not saved by *unexec*. Since the effects of calling the thread initialization routine are not idempotent, in many systems this thread initialization routine is automatically called by the start up code, only once. In our *C threads* [CD88] package, for example, a static flag is used to avoid initializing more than once. This static flag prevented the initialization routine from being called when we restarted from the *unexec* image. To resolve this problem, we forced the C start up code to call the initialization routine at the next start up.

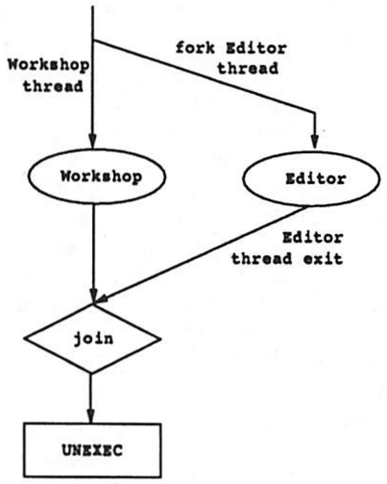


Figure 5: Synchronization on thread exit. To ensure that the Editor thread exits cleanly, the Workshop thread waits for the join operation to complete before proceeding to call *unexec*

Even if the thread initialization routine were being called every time, we still could not restart the dumped image of the program if it was not cleaned up properly before the unexec dump. The problem here was that the thread initialization routine tried to use the internal data structures for the threads already present in the dumped image from the previous run, but failed to re-initialize these structures corresponding to the states of the newly started threads. The stale data in these structures corrupted the threads and led to system crashes. To remedy this, we extended the thread-exit code to free all such data structures, forcing the thread initialization code to create new ones each time the program was restarted from an *unexec*. To ensure that the thread calling the unexec code was the last one to exit, we synchronized all thread exits via the *join* operation (Figure 5).

Sharing Memory

Having merged the threads, the Workshop and Editor now shared the same address space and could access each others' data directly. At this time, we no longer needed to keep separate copies of the data in both the Workshop and the Editor. Since the Workshop process does most of the computation with the data, we let the Workshop thread manage all the data. A transparent layer of memory-read/write primitives was implemented to perform object read and conversion from the Workshop format to that of the Editor (Figure 6).

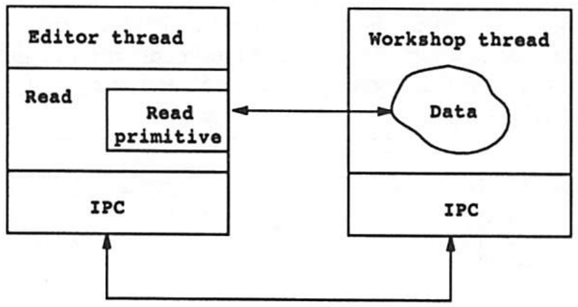


Figure 6: Data sharing: The new read primitive transparently reads the data directly from the Workshop heap and converting it to the format usable by the Editor, thus eliminating the need for IPC.

It is worth mentioning here that other implementations for this type of data sharing are possible, such as providing a separate thread to manage the entire shared object base, including synchronization primitives and memory management techniques (Figure 7). However, due to the architecture of *Matisse*, using a general data sharing method would have required us to rewrite many memory management operations and the garbage collector, which are already present as part of the Workshop. Designers of future systems should carefully consider exploiting the existing architecture before resorting to a more general scheme.

Garbage Collection Considerations

For an application with an embedded LISP environment, we must carefully consider the memory management issue. In *Matisse*, the Editor accesses Workshop's heap of data directly and must be protected from the Workshop's garbage collector during a critical time when it is holding pointers to Workshop's objects and reading them. This contention is handled by a simple mutex lock to be seized by either of these two entities as they try to get to the data. Locking can be done at a finer object level, but the infrequent Editor's look-ups make it costly and infeasible to implement a mutex for each object[2].

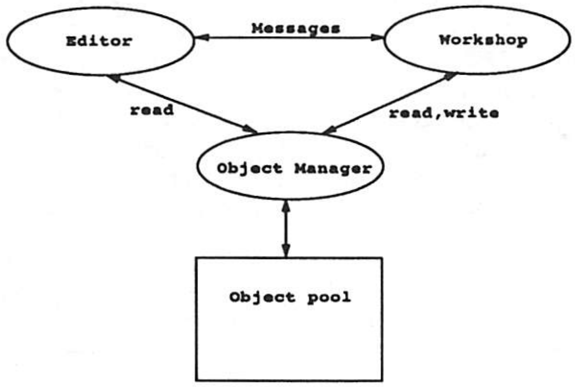


Figure 7: Shared-memory alternative: the object management mechanism can be implemented by an independent thread. Synchronization of object access between the Workshop and Editor threads can be handled by the Object Manager.

This simple locking scheme is sufficient if there is never a need for obtaining the mutex lock again during the critical section where the mutex lock is already held by either the garbage collector or the read primitive. However, in our system, there is a problem when the memory of the object heap runs low during the critical section of the Editor's read operation. In this case, the Editor thread would block by running the garbage collector, which would block waiting for the mutex lock to be released by the Editor. This deadlock is resolved by requiring the Editor to yield the mutex lock to the garbage collector and redo its read operation later[3].

Performance Evaluation

As described by the previous sections, *Matisse* evolved through three stages. Version 1 is a "straight port", featuring a one-to-one mapping of one UNIX process to one single-threaded task. Version 2 is the merge of these single-threaded tasks into one multi-threaded task with the IPC mechanism unaltered. Finally, version 3 is the multi-threaded version similar to version 2, but most of the IPC messages are replaced by direct read/write of data in shared memory between the threads. However, it was also necessary to implement version 3 in two steps: V_{3a} reduced the number of IPC bytes being transferred; and V_{3b} reduced both the number of IPC messages and bytes. Version 1 of *Matisse* serves as the baseline with which all performance measurements are compared[4].

Timing Measurements

To effectively illustrate the performance improvement from the re-architecture described in the previous section, two scenarios with significant IPC overhead were chosen as benchmark tests for each version of the system as it evolved from two single-threaded tasks to one multi-threaded task with shared-memory IPC. The IPC cost of these scenarios, mostly involving passing data back and forth between the Workshop and the Editor, is estimated at 40% of the system overhead. We use our *estimate* here because the actual execution time, comprised of *user time* [5] and *system time* [6], cannot be precisely measured across thread boundaries. The task of breaking a message into data packets, sending the packets across the wire, and reassembling the data stream, starts in one thread and ends in another.

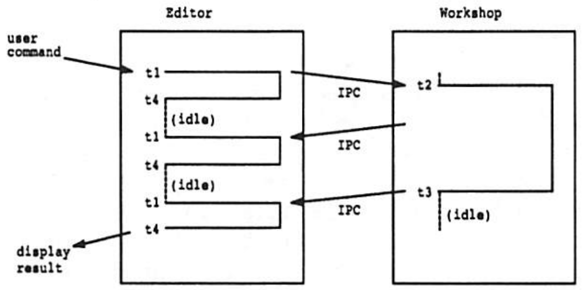


Figure 8: Timing measurement. This operation scenario contains 3 *Editor* transactions and 1 *Workshop* transaction. The total execution time is the sum of three (t4-t1) and one (t3-t2), in both user time and system time. Note that the idle time is not charged to the execution time of either thread.

The system timing utilities available to us could not be used to measure system execution time across thread boundaries. Thus, in order to coarsely measure the percentage of IPC overhead in a transaction, we computed, from time stamps, the *real* time it takes to send a message from the sending thread to the receiving thread. Since we can only measure this using *real* time, the number we get is, of course, slightly larger than the actual execution time of the two entities due to some other system threads (such as the scheduler, the window manager) utilizing the CPU in between. In our test machine,

[2]A thorough coverage of concurrent programming, threads, and mutexes can be found in [Bir89].

[3]The *redo* is done simply by having the read primitive release the lock, call the garbage collector, and then call itself with the original arguments.

[4]We made no attempt to compare the performance of the ported application to the original one running on the UNIX platform since there are simply too many system differences between the two operating systems to have a meaningful comparison.

[5]*user time* is the total amount of time the system spends executing in user mode

[6]*system time* is the total amount of time the system spends executing on behalf of the thread or process.

the experimental threads are the only user threads running other than the essential system threads. The CPU cost of the system threads, being constant across all measurements, does not affect the qualitative analysis of the performance profile, and amounts only to a small offset factor in the quantitative analysis. Along with the timing measurements, a pair of counters was also implemented to count the number of messages and the number of bytes being sent via the communication sockets.

The execution time of each scenario is measured by obtaining the total running time of all threads (or *tasks*, in version 1) between its start and end points. The acquisition of *user time* and *system time* is done by calling the system utility *getrusage*. With *getrusage*, we have a timing granularity of 1 microsecond, which is adequate for measuring transaction time lasting in the order of tens of seconds. The following paragraph and Figure 8 illustrate the timing measurement for a typical transaction.

Test Cases

The first test case is the startup sequence of actions that takes place as each user logs into *Matisse*. This scenario has a high percentage of IPC activities because the Editor and the Workshop must communicate with each other extensively to set up the environment for the user. The setup process involves the Editor getting the numerous program objects from the Workshop and initializing the display screen. This interprocess communication is done via sockets in versions 1 and 2, and via shared memory in version 3 of the system.

The second test case involves another IPC-intensive sequence of actions: modifying and saving a program object. In order to save a text object, the Editor first sends the modified object to the Workshop where it is validated, updated, stored, and sent back to the Editor to be displayed. In addition, the Workshop uses its rule base to determine and make the necessary changes in the system configuration.

Although the IPC overhead in both test cases is high (about 40%), they are slightly different in composition. The first test case involves numerous small IPC messages, while the second test case is comprised of fewer, but larger, IPC messages. This difference plays a key role in explaining the amount of system performance improvement and will be discussed in the following sections.

Light-Weight Threads And Shared Memory

The first set of experiments compares the performance of versions V_1, V_2, and V_{3a}. Table 1 illustrates the percentage reduction in IPC bytes and CPU time between every two versions compared.

As seen in Table 1 and Table 2, the improvement between V_1 and V_2 is indicative of the light-weight thread issues. Unfortunately, only a slight improvement is observed here because the scheduler of our operating system[7] does not provide light-weight threads with much advantage over conventional processes or tasks, and crossing the kernel boundary is expensive (as much as for processes). With a smarter scheduler, we would expect to do much better. For example, our system (HP9000 series 350) uses a virtual cache which can hold information for one address space at a time. A smarter scheduler could notice that the next thread running on the processor uses the same address space as the previous one, and avoid any unnecessary cache flush. Thus, by allowing the next thread to use valid data in the cache rather than causing expensive cache misses (after an unnecessary cache flush), an intelligent scheduler can cut the cost of a context switch on a virtual cache machine [CM89].

	Δ *IPC*	Δ *CPU*
$V_1 \rightarrow V_2$	-3.80%	-5.30%
$V_2 \rightarrow V_{3a}$	-31.25%	-11.11%
$V_1 \rightarrow V_{3a}$	-33.87%	-15.82%

Table 1: Performance Improvement, test Scenario 1: small IPC messages

	Δ *IPC*	Δ *CPU*
$V_1 \rightarrow V_2$	-1.87%	-5.05%
$V_2 \rightarrow V_{3a}$	-40.86%	-15.38%
$V_1 \rightarrow V_{3a}$	-41.97%	-17.11%

Table 2: Performance Improvement, test Scenario 2: large IPC messages

The tables also show that the performance improvement between V_2 and V_{3a}, however, is much more significant due to shared memory. By shifting from socket-based IPC to shared-memory IPC, we reduced the number of bytes to be sent via sockets by 31% and improved the overall system speed by 11%. Since roughly 40% of the original system over head is IPC related, we have effectively slashed the IPC overhead by approximately 25%.

In addition, second order effects exist which indirectly improve system performance. These beneficial factors occurred naturally as part of our re-architecture. For example, with the data sharing in version V_{3a}, the Editor no longer has to keep its local copy of the data, saving 400K of memory at run time. Being smaller in size, the merged version has less paging overhead, takes much less time to load into memory, and is less likely to be swapped out during a context switch.

[7]We used a University of Utah's port of MACH 2.0 on HP platform because it is readily available. Although MACH 2.5 exists for the HP9000 machines, it was not stable enough. OSF1 was not available at the time of this experiment.

Bytes vs. Messages

After examining V_{3a}, we discovered that we can do much better by not only reducing the number of bytes being sent between threads, but also the number of IPC messages. We examined the result of two slightly different variations of memory sharing techniques: version V_{3a} which reduces the IPC bytes, and version V_{3b} which reduces the number of IPC messages as well.

The original *Matisse* processes sent and received the [OID,Slot,Value] messages via sockets. Version V_{3a} eliminated most of the byte transfer by allowing the Workshop to send only the small [OID,Slot] messages, while making the Editor perform the lookup and copy of large *Value* data fields directly from the Workshop's memory.

Version V_{3b}, reduced the number of messages to only one per object. The Editor thus had a greater responsibility to look up the Slot and Value attributes of the desired object. The additional computation needed to determine what slots of the given OID needed updating was still much cheaper than sending the list of [OID,Slot] via IPC messages. Since most of the IPC messages are small, and the cost of sending messages up to a certain size is constant, the benefit is not fully gained if we just reduced the IPC bytes. For example, in our system, the cost is the same for messages up to 8K bytes in size, so it was not very beneficial to just reduce the size of data packets since the payoff would stop after the 8K-bytes packet size.

	Δ *IPC bytes*	Δ *IPC msgs*	Δ *CPU*
$V_1 \rightarrow V_2$	-3.80%	0%	-5.30%
$V_2 \rightarrow V_{3a}$	-31.25%	0%	-11.11%
$V_1 \rightarrow V_{3a}$	-33.87%	0%	-15.82%
$V_{3a} \rightarrow V_{3b}$	-80.88%	-80.50%	-28.50%
$V_1 \rightarrow V_{3b}$	-87.35%	-80.50%	-39.81%

Table 3: Performance Profile, test Scenario 1: small IPC messages

	Δ *IPC bytes*	Δ *IPC msgs*	Δ *CPU*
$V_1 \rightarrow V_2$	-1.87%	0%	-5.05%
$V_2 \rightarrow V_{3a}$	-40.86%	0%	-15.38%
$V_1 \rightarrow V_{3a}$	-41.97%	0%	-17.11%
$V_{3a} \rightarrow V_{3b}$	-85.23%	-82.95%	-20.44%
$V_1 \rightarrow V_{3b}$	-91.42%	-82.95%	-34.05%

Table 4: Performance Profile, test Scenario 2: large IPC messages

Additional performance is gained by reducing also the number of messages. This explains the observation that, in test Scenario 1 where the communication activities involve many small messages, a reduction of 80.88% of IPC bytes and 80.50% of IPC messages reduced the overall CPU time by 28.50%. And in the test Scenario 2 where the communication pattern is comprised of fewer but larger messages, a comparable reduction of 85.23% of IPC bytes and 82.95% of IPC messages led to a much smaller reduction of 20.44% overall CPU time.

The performance data in Table 3 and Table 4 show that, in version V_{3b}, we have reduced the traffic volume by as much as 85% and CPU time by 28% over version V_{3a}. Comparing this performance with the original version V_1, we have achieved approximately 40% reduction in CPU time in running our set of test cases. This reduction means we have virtually eliminated the original system overhead related to IPC, which was 40%. Obviously, there is still a trickle of IPC messages present in the system since we have not completely eliminated it yet[8] , but the performance improvement resulting from any secondary effects have already covered this cost. In addition, the performance gained from these secondary effects also covered the cost incurred from the shared-memory read/write protocol.

System Threads vs. User Threads

The concept of merging single-threaded processes to gain shared-memory communication capability can be applied to coroutine packages as well. Such an approach does not require a migration to an operating system with multi-threaded support, but rather the reduction of processes to *user threads* of a coroutine package. If the user already has a coroutine package that manages the scheduling of user threads, the merge can yield performance improvement by the resulting user threads having shared-memory IPC. An important difference between user and kernel threads is that in a multi-threaded multiprocessor environment, the system threads can potentially be scheduled on several processors, exploiting the real concurrency, while the user threads in a coroutine package can be scheduled and run one thread at a time, on only one processor at a time.

Multiprocessor Implications

A logical next step would be to rehost *Matisse* onto a multi-threaded, multiprocessor environment. The migration to this type of environment would require no additional work beyond that described. In a multiprocessor environment, each thread is a candidate for scheduling on any processor. If two or more threads sharing memory are run concurrently on different processors, the system transparently manages the shared memory addressed by the

[8]For this experiment in process migration, we decided to only implement the shared memory IPC to replace socket-based data transfer between the two threads. Some socket-based IPC messages are still used for control and communication between the threads. These messages can also be replaced by signals and condition variables between threads, with control data transfer facilitated by common data buffers with locks.

different CPUs. Given that the operating system for the multiprocessor machine is designed and implemented properly, we would expect to see a multi-threaded application run faster on a multiprocessor machine than on a uniprocessor machine as a result of the parallelism of the multiprocessor architecture.

To better take advantage of multiprocessor machines, however, this work can be extended to break up the code into many threads, hence "parallelize" the application whenever possible. The numerous threads can then be run in parallel across the processors, thereby effectively utilizing the hardware resources. For example, in *Matisse*, the Workshop's garbage collector can be implemented as a separate thread. Also, object updates or queries can be done in parallel by forking a thread for each job, rather then simply doing them serially.

Porting Effort

The port and re-architecture of this experiment took about two man-months complete, since we had no previous experience in porting a system this large and complex. The effort was eased considerably by the help and insight of people in the labs who had ported the MACH operating system onto our machines. Along the way, we carefully documented our path, as reported above, so that future porting efforts can be done much faster. Knowing what we know now, this experiment can be repeated in a one or two-weeks time frame. The approximate time we spent on each part of this experiment is listed in Table 5.

Porting Task	*Time Taken*
UNIX→MACH port	2 weeks
Single-threaded → Multi-threaded	1 week
I/O contention problems	1 week
Unexec dump problems	2 weeks
Shared-memory IPC implementation	2 weeks

Table 5: Experiment time table

The Importance Of Threads

Earlier sections of this paper on the design and porting effort of Matisse describe at length the threads mechanism. The performance charts, however, show that most of the performance gained is from shared-memory, not threads. The importance of threads cannot be under-estimated because it is the mechanism which delivers shared-memory. Without threads, there is no way to obtain the performance improvement the way we had with shared-memory and still keep the existing program interface on IPC. Changing this interface to use system V shared memory would require months of rewrite for a software system this complex. Although the system threads themselves do not contribute much toward the performance gain in this particular system, they are getting faster and lighter, as the importance of threads increase now and in the future, with the prevalent needs for multiprocessing and fine-grained parallelism.

Conclusion

The result of the experiment matched our initial expectations and we have formulated a rough guideline for the migration process. By laying out the steps and identifying the potential problems associated with each step, we hope future migrations can be done quickly and painlessly. However, not all components of a multi-process application should be merged as concurrent threads. Although migrating single-threaded processes to concurrent threads within a multi-threaded task enables the threads to communicate cheaply via shared-memory, the trade-off is that we lose the machine-independent property of the original processes. By porting and merging processes as concurrent threads, they must now be executed on the same machine, whereas they previously could be run on different machines. Thus, candidates of this re-architecture process should be those that have a large IPC overhead, and always reside on the same machine. Many existing applications fit this requirement, and are good candidates for migration.

Acknowledgments

The authors would like to thank professor Jerome H. Saltzer of MIT for the insightful advice, questions, and comments about the project.

Our colleagues Scott Marovich and Debbie Caswell provided enormous help and technical expertise with the UNIX and MACH/OSF1 operating systems. Brian Beach assisted with the memory management issues of the C and LISP environments. Many thanks to Debbie, Kevin Wentzel, and Professor Saltzer for their suggestions on improving the paper.

We thank Dan Fishman, Martin Griss, and Paula Hawthorn for their continuous support of the Matisse project under which the experiment was conducted.

References

[ABB+86] M. J. Acetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new Kernel Foundation for UNIX Development. Proceedings of Summer Usenix, page 5, July 1986.

[Bir89] Andrew D. Birrell. An Introduction to Programming with Threads. Technical report, Digital SRC, January 1989.

[CD88] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, School of Computer Science, pages 1-10, June 1988.

[Cle88] Geoff Clemm. The Workshop System: A practical Knowledge-Based Software

Environment. Proceedings of the 3rd ACM Software Engineering Environments Conference, Pages 55-64, December 1988.

[CM89] D.L. Caswell and S. Marovich. STL MACH Project Retrospective. Technical Report STL-89-20, Hewlett-Packard Laboratories, Software Systems Lab, August 1989.

[FR86] R. Fitzgerald and R. F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. ACM Transactions on Computer Systems, 4(2):147-149, May 1986.

[JRG+87] A. Tevanian Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. Mach Threads and the UNIX Kernel: The Battle for Control. Technical Report CMU-CS-87-149, Carnegie-Mellon University, School of Computer Science, August 1987.

[Mar90] Scott B. Marovich. Intraprocess Concurrency under UNIX. Technical Report HPL-91-02, Hewlett-Packard Laboratories, March 1990.

[Sal66] Jerome H. Saltzer. Traffic Control in a Multiplexed Computer System. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, page 2, June 1966.

[Sta84] R. Stallman. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. In D. R. Barstow, H. E. Shrobe, and E. Sandelwall, editor, Interactive Programming Environments, pages 300-325. McGraw-Hill Book Company, 1984.

[TR87] A. Tevanian and R. F. Rashid. Mach: A Basis for Future UNIX Development. Technical Report CMU-CS-87-139, Carnegie-Mellon University, School of Computer Science, pages 1-2, June 1987.

Author Information

Thuan Pham is a Member of Technical Staff at Hewlett-Packard Laboratories, Software Technology Lab. He received his M.S. in Electrical Engineering and Computer Science and the B.S. in Computer Science and Engineering from the Massachusetts Institute of Technology. His research interests include software engineering, operating systems, and user interface. Contact him at pham@hplabs.hp.com.

Pankaj Garg is a Project Leader at Hewlett-Packard Laboratories, Software Technology Lab. He got his Ph.D in Computer Science at the University of Southern California, in February 1989. Pankaj was an All-University-Pre-Doctoral-Merit fellow of the graduate school, USC, for the years 1984 through 1987. He was a research associate in the Computer Science Department, USC, from 1987 through February 1989. He got his Bachelor of Technology degree in Computer Science, from Indian Institute of Technology, Kanpur (INDIA). His research interests are in artificial intelligence, hypertext systems, and software engineering. Contact him at garg@hplabs.hp.com.

Cheap Mutual Exclusion

William Moran, Jr. – Swiss Bank Corp Investment Banking[1]
Farnam Jahanian – IBM T. J. Watson Research Center

ABSTRACT

A new method of enforcing mutual exclusion among concurrent processes on uni-processors running *UNIX* is presented in this paper. When a process attempts to obtain a lock, no race condition will occur unless the process is preempted. The central idea is that a process can avoid a race condition if preemption is made visible to the process when it is rescheduled. Two possible implementations of this idea are discussed in depth. The proposed solutions do not require special hardware support or disabling of interrupts during a critical section.

1. Introduction

This paper presents an alternative approach to mutual exclusion on a uni-processor *UNIX* system. The motivation for this work arises from the expense frequently associated with achieving mutual exclusion among concurrent processes. On many *UNIX* systems, such protection requires several system calls per lock access; the use of semaphores[2] to protect shared resources is frequently impractical due to the expense. A common use of shared memory is for storage of data structures accessed by several processes; the contention for these data structures is infrequent enough that costly locking schemes are wasteful. On many systems, the use of semaphores requires a system call to lock the resource; another system call is required to unlock the resource. Even when there is no contention for the lock, both system calls are executed. These system calls may be very expensive depending on how much support the hardware provides, but even in the best case, system calls are expensive due to the context switches and the change from user mode to kernel mode. When there is contention among processes in accessing shared resources, it is often the case that the actual code for manipulating shared data structures is only a few lines. Hence, the expense associated with obtaining a lock is disproportionate to the actual use of the shared resource. The overhead of obtaining and releasing a lock is particularly important if a process accesses locks multiple times.

Some hardware provides support for a test and set or any of a number of similar sorts of constructs; on architectures which support such atomic operations, the Intel i486 for example, locking is trivial. However, not all architectures provide support for this sort of atomic operation; For example, the IBM RS/6000 processor intentionally does not provide any such instruction. However, there is a special operation called **CS**. This implements the compare and swap instruction as a pseudo system call; while this avoids the overhead of a system call, it requires both special hardware support and kernel code. So, while this is a compromise between having a special atomic hardware instruction (the hardware support required for the **CS** instruction is somewhat more general in nature, i.e., it is useful for other things) and having no support, it is not possible to use this technique on most other machines [7,6,8].

This paper presents a set of practical solutions to this problem. The solutions presented here are general enough that they could be used on many machines running *UNIX* ; in particular, it will work on any machine with POSIX signals or their ancestor, BSD signals. The proposed solutions do not require special hardware support or disabling of the interrupts while a lock is being acquired; the solutions presented here do require that the Operating System kernel be modified. The characteristics desired of such solutions are that they should be very inexpensive in the ordinary case (no contention), and they should be no more expensive than semaphores in the presence of contention. Ideally, the solution should require no more user code than the standard semaphore example:[3]

```
int sem_id;
...
key2 = ftok( FTOK2_FILE, FTOK2_ID );
sem_id = semget( key2, 1,
                 IPC_CREAT | 0777 );
semctl( sem_id, SEM, SETVAL, 1 );
...
p(sem_id);
/* critical section */
v(sem_id);
```

Code that uses semaphores typically requires 3 system calls to initialize the semaphore, and each

[1]William Moran was with IBM T. J. Watson Research Center when this work was performed.

[2]Semaphores means System V style semaphores in this context.

[3]This uses the Sun implementation of standard System V UNIX IPC operations.

acquisition and subsequent freeing of the semaphore requires 2 system calls. Since system calls typically have path lengths of at least 3000 instructions, system calls are considered expensive. In particular, if a program makes many lock accesses, the cost associated with getting the locks becomes extremely important.

The remainder of this paper is organized as follows: The next section discusses an overview of the proposed approach for achieving mutual exclusion in *UNIX* systems. Section 3 and Section 4 present two different implementations of this approach. Section 5 is a discussion of a few of the subtler implementation issues. The last section contains concluding remarks.

2. Approach

Mutual exclusion is needed to avoid race conditions associated with the modification of shared data by concurrent processes [4,5,2,3,11,14] The problem of avoiding a race condition can be formulated in an abstract way as imposing a shared lock (e.g., shared variable) on access of a critical section. In a uniprocessor *UNIX* system, obtaining a lock may be non-atomic if the process attempting to obtain the lock is preempted while it is obtaining the lock. For example, suppose a process obtains a lock by writing its process id into a shared variable, called `shared_lock`. This variable is initially set to zero to denote that the lock is free.

```
if (shared_lock == 0)
  shared_lock = my_pid;
else ...
```

If the process is preempted after testing `shared_lock` but before writing its process id into the shared variable, another process may obtain the lock by executing the same sequence of instructions (without preemption). When the first process is rescheduled at the point of preemption, it will acquire the lock while it is being held by the second process.

In a uni-processor system, if the sequence of instructions to obtain a lock is executed without preemption by the kernel, no race condition can occur. This simple observation is very important: if a process obtains a lock by writing its process id in the lock, then if it is not preempted while so doing, the lock can be held by only one process. However, if the process is preempted while obtaining a lock, the race condition may be avoided by informing the process that it was preempted. The central idea here is very simple: make preemption visible to processes that require the service. If a process attempting to lock a shared variable is preempted, it is notified when rescheduled so that another attempt at acquiring the lock is made, avoiding the race condition. The two subsequent sections describe alternative implementations of this concept in a *UNIX* system.

3. First Version

The first solution is intended to show the viability of achieving mutual exclusion by making process preemption visible to the process after it is rescheduled. The proposed solution involves asynchronous notification of a user process when it has been preempted. The signal facility, supported in all UNIX-like systems, is used for asynchronous notification. The process receives a signal when it is rescheduled after the preemption; this signal indicates that a preemption occurred. The key point is that the process returns to the signal handler rather than to the point at which the preemption occurred. In the signal handler, the process can be forced to resume execution at a point where another attempt at acquiring the lock can be made. For example:

```
jmp_buf context;
int ret;
...
void sig_handler(int num)
{
  longjmp(context,0);
}
...
setjmp(context);  ← execution resumes here
statement1;
statement2;       ← preemption occurs here
statement3;
statement4;
...
statementn;
```

The above code segment invokes two library procedures: `setjmp` and `longjmp`. The `setjmp` call saves the context of the process at the point of invocation. The process context includes information such as the values of the registers and the PC. The `longjmp` call , when invoked inside the signal handler, will cause the process to resume execution at the point where the setjmp was called. If a signal gets delivered to this code upon preemption, and the signal is caught by `sig_handler`, then execution will resume at the setjmp rather than where the code was preempted.

The preceding paragraph illustrated how normal execution of a process can be altered via the signal facility. This forms that basis for the first solution to making preemption visible to a process. Specifically, the following code implements mutual exclusion if the kernel has been modified to allow the preemption signal; specifically, it is necessary that the preemption signal handler be the first code executed upon return from a preemption[4,5]. The

[4]This code uses the AIX V 3 implementation of POSIX signals and signal handlers.

[5]This and subsequent examples assume that `sleep()` causes a process to be preempted automatically.

following code uses a new system call, `preempt_sig` which takes as an argument, the number of the signal which is to be sent to the process when it returns after being preempted. The subsequent examples in this paper will use signal number 43. A system without the extended signals added by POSIX could simply use one of the signals not normally used anymore. See Figure 1.

This code works as follows: it gets the process id of this process (1). The process id is an integer that uniquely identifies every process. The next two lines (2 & 3) define two signal handlers used for the preemption signal. The first of these will simply ignore the signal; as the result of *UNIX* semantics, ignored signals are never delivered [9,10], so while `a` is the specified handler, preemption will have no extra cost. While the handler bound to `b` is in effect, `sig_handler` will be called upon delivery of the preemption signal. (4) simply causes the preemption signal to be ignored. (5) is a new system call that tells the kernel that this process should have the specified signal, in this case 43, delivered upon preemption. The critical section follows this; the `setjmp` call saves the state of the process at this point. The `longjmp` in the signal handler will cause execution to be resumed here. (7) rebinds the preemption signal to `sig_handler`, so once this statement has been executed, the process, upon being preempted, will resume execution in `sig_handler` rather than at the point of preemption. (8-11) are the standard method of acquiring a shared lock. Finally, (12) turns off the preemption signal. Figure 2 shows what happens upon preemption. The idea here is that if, on a uniprocessor, the statements (8-11) execute atomically, then we are assured that the lock has been acquired by only one process. However, if preemption occurs anywhere in these statements, then it is possible that another process has come in and acquired the lock, so we need to check the lock to see if this has happened. If it has, we force preemption with the `nsleep` call (`nsleep()` does the same thing as usleep except in nano-seconds). There is one important assumption in this code; (11) must not be capable of leaving the variable `shared_lock` in a corrupt state, but on every architecture of which we are aware, preemption

```
#define SIG_NUM 43
jmp_buf context;
int ret;
pid_t ourpid;
struct sigaction a,b;
...
void sig_handler(int num)
{
   longjmp(context,0);
}
...
ourpid = getpid();                                          (1)
a.sa_handler = SIG_IGN;                                     (2)
b.sa_handler = sig_handler;                                 (3)
...

sigaction(SIG_NUM,&a,NULL);                                 (4)
/* The following instructions get the lock */
preempt_sig(SIG_NUM);                                       (5)
setjmp(context);                                            (6)
sigaction(SIG_NUM,&b,NULL);                                 (7)
if ((shared_lock != 0) && (shared_lock != ourpid))          (8)
   nsleep(1);                                               (9)
else                                                       (10)
   shared_lock = ourpid;                                   (11)
/* We have the lock, so we turn off the signal */
sigaction(SIG_NUM,&a,NULL);                                (12)
/* critical section */
...
/* Free the lock */
shared_lock = 0;
```

Figure 1: Locking

during this statement either results in `shared_lock` being unchanged, or in the assignment actually succeeding. On an architecture where this assignment could result in `shared_lock` changing to something other than `ourpid`, this method of mutual exclusion will not work.

The performance of this code is about 25% better than the comparable code using semaphores when run on an RS/6000; based on the performance of semaphore operations and signal operations, it seems as if this code would be 50% as costly as the equivalent semaphore code on a Sun 4 running SunOS 4.0.3.

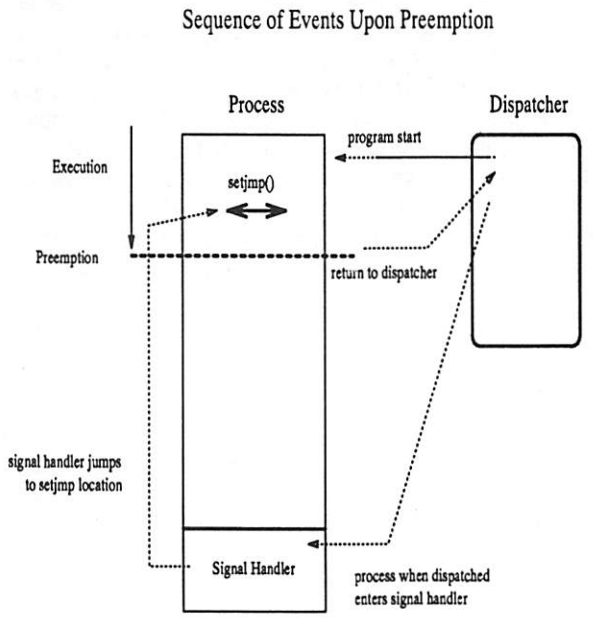

Sequence of Events Upon Preemption
Process
Dispatcher
Execution
program start
setjmp()
Preemption
return to dispatcher
signal handler jumps to setjmp location
Signal Handler
process when dispatched enters signal handler

Figure 2: Preemption Events

Significant performance improvement can be obtained when several locks are being acquired and freed in one section of code. Specifically, it is possible to leave the signal delivery turned on and simply check in the signal handler whether it is necessary to perform the `longjmp`. After the first lock access, the cost of obtaining the subsequent locks is reduced to the cost of invoking `setjmp` to save the process context. The cost of releasing a lock is a single assignment. However, there is a small overhead in invoking the signal handler after the process is rescheduled. The code for obtaining and releasing several locks looks something like Figure 3.

The code segment enables the preempt signal before an attempt to acquire any lock is made. The process context is saved and a local variable `should_jump` is set prior to each lock access. If the process is preempted while acquiring the lock, the variable `should_jump` is tested inside the signal handler and the process is forced to make another attempt to obtain the lock. After obtaining the lock, the variable `should_jump` is reset. This is an effective solution since returning from a preemption into a signal handler does not slow the code down excessively. This code should be substantially faster than the equivalent semaphore code; it saves two system calls per critical section, i.e., both `sigaction` calls, but it has not been tested enough to state this definitively.

```
#define SIG_NUM 43
jmp_buf context;
int ret;
pid_t ourpid;
struct sigaction a, b;
int should_jump = 0;
...
void sig_handler(int num)
{
   if (should_jump)
      longjmp(context,0);
   else
      return;
}
...
ourpid = getpid();
a.sa_handler = SIG_IGN;
b.sa_handler = sig_handler;
...
sigaction(SIG_NUM,&a,NULL);
preempt_sig(SIG_NUM);
setjmp(context);
sigaction(SIG_NUM,&b,NULL);
should_jump = 1;
if ((shared_lock != 0) &&
          (shared_lock != ourpid))
   nsleep(1);
else
   shared_lock = ourpid;
should_jump = 0;
...
setjmp(context);
should_jump = 1;
... /* acquire lock */
should_jump = 0;
...
setjmp(context);
should_jump = 1;
... /* acquire lock */
should_jump = 0;
sigaction(SIG_NUM,&a,NULL);
```

Figure 3: Obtaining and releasing locks

This section has presented code that uses the visibility of preemption as a way of ensuring mutual exclusion. The code presented here uses this feature in a rather naive fashion, but it shows the power of this idea. The final example in this section demonstrates one of the most important aspects of this idea: by making locks very cheap to obtain, this code makes it possible to write code that shares many resources. Since serial access to these resources can be insured cheaply, the cost of sharing resources can be dramatically reduced.

4. Second Version

In the previous section we presented a solution that uses the preempt signal in the obvious way. While acquiring a lock, the process enables the preempt signal so that it is notified of a preemption after being rescheduled. The preempt signal is disabled after the lock is obtained. As illustrated in the preceding section, the cost of enabling/disabling the signal can be amortized among multiple lock accesses. However, a solution which reduces this overhead is desirable. In this section, we present a solution that uses the signal in a slightly more subtle fashion. If we modify the system call

`preempt_sig`, that we presented in the previous section, so that it takes the address of a flag in user space, then the kernel can check this flag to determine whether to send a signal. In this case, protecting the acquisition of the lock will be the cost of setting this variable. For example:

```
#define SIG_NUM 43
jmp_buf context;
int flag = 0;
int ret;
pid_t ourpid;
struct sigaction a,b;
...
void sig_handler(int num)
{
   if (flag)
    longjmp(context,0);
}
...
ourpid = getpid();
b.sa_handler = sig_handler;
...
sigaction(SIG_NUM,&b,NULL);
preempt_sig(SIG_NUM,&flag);        (1)
/* Code to get a lock */
setjmp(context);
flag = 1;                          (2)
if ((shared_lock != 0) &&
           (shared_lock != ourpid))
    nsleep(1);
else
    shared_lock = ourpid;          (3)
/* We have lock;
           turn off the signal */
flag = 0;
/* critical section */
...
shared_lock = 0;   /* Free the lock */
```

This version uses the new system call (1) that takes the address of `flag`, and it simply sets `flag` to 1 when the preempt signal should be delivered. When the signal is not necessary, all that is needed is to set the flag to 0. Compare the critical section here (the code between (2) and (3)) with that above. Two system calls have been eliminated. As a result, we are not using any system calls to achieve the necessary protection. There are only two costs here; the first is the `setjmp` and the `longjmp` if preemption occurs, but this is very small. The other cost is slightly harder to quantify, since it is incurred in the kernel. Understanding this cost requires that the code in the kernel be understood. These costs compare favorably to the semaphore code. The cost incurred when a process blocks on a semaphore is fairly high.

Kernel code is required in two places, the first is the system call `preempt_sig`. This code needs to set a flag associated with the process that indicates that the process should be signalled when preempted; this code also needs to store the number of the signal to send. These two quantities require 7 bits (1 for the flag and 6 for the signal number); the pointer to the flag requires 32 bits, so 39 bits are associated with the process. Finally, because the flag, which resides in the users address space, is going to be checked in the kernel, the page containing this flag must be pinned into real memory; since this flag will be checked in the dispatcher, and the code in this part of the kernel is not allowed to page fault, the memory access to check this flag must not page fault. This page is pinned with the `pinu` kernel service. Unless the system is swapping (as opposed to paging), this appears to have minimal impact. Thus, the process will have one page that is always in memory.

The second place where kernel code is required is in the dispatcher; after the dispatcher has determined which process will be run next, it is necessary to check the flag that indicates whether the process should be signalled. The code to do this is something like:

```
if (p->presig)
   if (*(p->sigflagaddr))
      pidsig(p->pid,p->sigtosend);
```

where the first `if` checks to see whether this process should ever receive the preempt signal; the second `if` checks whether the process should have the signal delivered at present (`sigflagaddr` is a pointer to user address space), and the `pidsig` call sends the appropriate signal to the process.

This approach is clearly superior to the first approach presented. It saves two system calls per lock, and the code the user needs to understand is considerably simpler. This code compares very favorably with that required in the semaphore case; it is much faster (obtaining the lock is very inexpensive), and the number of system calls required for the setup is the same (3 in each case). These system calls are also slightly cheaper than those used for semaphores. The mechanism required by this approach has been implemented and tested on an RS/6000 running AIX 3.1, and the added cost in the kernel is not measurable. This code also has the property that the complexity (as the user sees it), is no worse than for semaphores. This solution attains all the qualities that were originally hoped for. The complexity of use is slightly better than for semaphores. The performance when there is no contention for the lock is substantially better than equivalent code that uses semaphores. Finally, the performance is no worse when the lock is already held by another process; in both cases, the process blocks. The only extra cost for this code occurs when the process is preempted while obtaining the lock; this extra cost is dwarfed by the reduced cost in the ordinary case. Due to these advantages, this method of achieving mutual exclusion seems preferable to the use of semaphores.

5. Implementation Notes

The astute reader will have noticed several potential problems with the solutions presented above; this section attempts to address these. In particular, the following might be seen as potential problems: nesting of signal handlers, starvation as a result of repeated preemption, and hidden assumptions as to atomicity of operations. This section will show why each of these fails to be a problem, and a brief explanation of each is included.

UNIX makes few guarantees about what can safely be executed within a signal handler. In particular, in the case of nested signal handlers, it is the case that almost anything executed can cause problems. However, *POSIX* 1003.1 [9] makes the guarantee that `longjmp` can be executed from *a non-nested* signal handler. Since it is the case, that the signal handlers here will not be nested, the `longjmp()` should be safe. It will also be noted that this code assumes that `nsleep(1)` will force preemption. It is certain that `nsleep` with some value will force preemption, and an incremental backoff could be implemented as `nsleep(i*=2)` where `i` had some suitable initial value.

The above code contains the possibilty of starvation if the process has few enough pages. If, upon return from the signal handler, the process page faults, then the process will immediately return to the signal handler. As a result, it might be necessary that the process have two pages available to it; one might be used to hold the signal handler code, and the other would hold the code where the `setjmp` was executed. If the code to obtain the lock spans multiple pages, a similar problem may arise; it should be noted that this is only a problem if the process can obtain only one page. However, a process that cannot acquire two pages is likely to have many other problems as well, so this seems like a fairly minor drawback.

An interesting example arises when a process is preempted twice while trying to get the same lock. In particular, a problem might be thought to arise if a process is preempted in the signal handler for the preemption signal; the potential problem occurs since the second instance of the signal might not be delivered until the next trip through the kernel. Consider a process in the preempt signal handler that is preempted. When the process resumes, the new instance of the signal will not be delivered since the signal is currently being handled; the second instance of the signal will be pending. The second instance of the signal will not be received until the next time the kernel gets control. In the first example, since the instruction executed after the `setjmp` is a system call eg `sigaction`, the signal will be delivered immediately, the process will jump back to the same place, and it will continue. In the second example, since no system call is executed, the signal will be delivered some arbitrary time later. However, the `longjmp` will only occur if the flag is still set, and the correct semantics are preserved because the lock cannot be corrupted. Further, since the order of operations is: `setjmp`, flag = 1, even if the signal gets delivered at the time of the next lock use, the correct context would be used. Of course, this sequence of events is the least favorable, but it is handled correctly.

Finally, as noted above, the assumption is made that an assignment of a wordsize quantity to a word size location cannot be preempted in such a way as to leave the location with a value other than either its initial value or the value being assigned. Simply put, if the statement `a = 7` is executed with `a` having an initial value of zero, `a` cannot have a value other than zero or seven no matter what the operating system may do (other than crash the machine possibly). The reason for this restriction should be obvious, and we know of no machines that do not obey this restriction. A failure to provide this facility would make the solutions presented here invalid; however, a machine that acted this way would need to have a multi instruction assignment, and this seems untenable.

6. Conclusion

The cost of obtaining and releasing a lock is particularly important when concurrent processes require mutual exclusion for accessing shared data structures stored in memory. In such cases, the duration of a critical section is often very short, and expensive methods for acquiring locks impose unnecessary overhead on those processes which manipulate the shared data structures. The characteristics desired of a method of achieving mutual exclusion on a *UNIX* system are that it should be simple for a program to use, it should be extremely inexpensive to use when there is no contention, and it should be as cheap as possible when contention exists. The two solutions presented here can be made as simple to use as semaphores, so they meet the first criteria. When there is no contention for the lock, and the process does not get preempted, the second method presented is essentially free. Even in the presence of contention, the second method presented is extremely inexpensive. Since much of the cost of using a semaphore is due to the overhead of making a system call, the proposed solution is considerably less expensive than semaphores on machines without hardware support. Furthermore, it is not too much more expensive, in the worst case, than semaphores even with hardware support. Combining ease of use and inexpensive operation as it does, this method of achieving mutual exclusion should make shared resources practical in many cases where they have been considered too expensive.

7. Bibliography

[1] Maurice J. Bach.*The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[2] P.Brinch Hansen. *Operating System Priciples*. Prentice-Hall, Englewood Cliffs, N.J., 1973

[3] P.Brinch Hansen. The programming language concurrent pascal. *IEEE Transactions on SE*, SE-1(2):199-207, June 1975.

[4] E. W. Dijkstra. Co-operating sequential processes. In F.Genuys, editor, *Programming Languages*. Academic Press, London, 1968.

[5] C. A. R. Hoare. Monitors: An operating system concept. *Communications of the ACM*, pages 549-557, October 1974.

[6] IBM Corp. *AIX Version 3 for the RS/6000 Calls and Subroutines Reference: Base Operating System Volume 2*, first edition, March 1990.

[7] IBM Corp. *AIX Version 3 for the RS/6000 Calls and Subroutines Reference: Base Operating System Volume 1*, first edition, March 1990.

[8] IBM Corp. *AIX Version 3 for the RS/6000 Calls and Subroutines Reference: Kernel Volume 5*, first edition, March 1990.

[9] IEEE, New York, New York. *POSIX 1003.1*, 1989.

[10] Samuel Leffler, Marshall Kirk McKusick, Michael Karels, and John Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, Reading, Massachusets, 1989.

[11] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, pages 115-116, June 1981.

[12] Marc J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[13] SUN Microsystems Inc. *SUN OS 4 System Services Overview*, revision a of 9 may 1988 edition.

[14] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

Author Information

William Moran has BS and MS degrees in CS from Union College. He joined IBM Research in 1988 after having been a PhD student at Yale. At IBM Research he worked on real-time and distributed programming languages, real-time systems, distributed and fault-tolerant systems, and a juggling robot. He left IBM Research in 1992 to join a new proprietary trading group at Swiss Bank Corp. Investment Banking as a Senior Computer Scientist. Reach him via US Mail at SBCI; 4th Floor; 222 Broadway; NY, NY 10038. Reach him electronically at wlm@panix.com.

Farnam Jahanian received a Ph.D. in Computer Science from the University of Texas at Austin in 1989. He is currently a Research Staff Member at the IBM T.J. Watson Research Center. His reasearch interests include specification and analysis of real-time systems and fault-tolerant computing. His address is: IBM T.J. Watson Research Center; P.O. Box 704; Yorktown Heights, NY 10598. Contact him by e-mail at farnam@WATSON.IBM.COM.

TDBM: A DBM Library With Atomic Transactions

Barry Brachman, Gerald Neufeld – Dept. of Computer Science, University of British Columbia

ABSTRACT

The dbm database library [1] introduced disk-based extensible hashing to UNIX. The library consists of functions to use a simple database consisting of key/value pairs. A number of work-alikes have been developed, offering additional features [5] and free source code [14,25]. Recently, a new package was developed that also offers improved performance [19]. None of these implementations, however, provide fault-tolerant behaviour.

In many applications, a single high-level operation may cause many database items to be updated, created, or deleted. If the application crashes while processing the operation, the database could be left in an inconsistent state. Current versions of dbm do not handle this problem. Existing dbm implementations do not support concurrent access, even though the use of lightweight processes in a UNIX environment is growing. To address these deficiencies, tdbm was developed. Tdbm is a transaction processing database with a dbm-like interface. It provides nested atomic transactions, volatile and persistent databases, and support for very large objects and distributed operation.

This paper describes the design and implementation of tdbm and examines its performance.

1. Introduction

In the UNIX environment, the dbm database library[1] [1] has become widely used to provide disk-based extensible hashing for a variety of applications. The library consists of functions to use a simple database consisting of *items* (key/value pairs). A number of work-alikes have been developed, offering additional features [5] and free source code [14,25]. Recently, a new package was developed that also offers improved performance [19] and there are plans to add a transaction mechanism to this package [20].

As an integral part of our distributed system research, an efficient and reliable database was required. In these and many other applications, a single high-level operation may result in several objects being updated, created, or deleted. If the application or host system crashes while processing the operation, the database must not be left in an inconsistent state.

Many distributed applications have a server component that can handle many client requests simultaneously. For example, in the case of the X.500 Directory Service [4], a server called the Directory System Agent is most naturally implemented as a multi-threaded application, with one or more threads servicing each client request. To maximize the level of concurrency, the database should permit simultaneous read-only and update operations while guarding the database against inconsistencies.

Current versions of dbm, however, do not meet the requirements of these types of applications. Most importantly, they do not guarantee consistency in the face of crashes. Existing dbm implementations cannot be used in a multi-threaded application, even though the use of lightweight processes in a UNIX environment is growing. Also, no assistance for implementing distributed and replicated databases is given.

To meet these requirements, tdbm (dbm with transactions) was developed. Tdbm provides nested atomic transactions [13], volatile and persistent databases, support for very large data, stores the database within a single UNIX file, and provides assistance for managing distributed databases. Tdbm can be configured to operate either as a conventional UNIX library or as part of a multi-threaded application. The EAN object store [17], used by the EAN X.500 directory service [16], is based on tdbm.

In the next section, the major design decisions associated with tdbm are examined. In Section 3, we look at the implementation of tdbm and in Section 4 an evaluation of the performance of tdbm is given. Finally, the paper concludes with some thoughts about our experiences with tdbm and possible extensions and improvements. The manual page for the library appears in the appendix.

[1]This work was made possible by a grant from OSIware, Inc.

2. Design

In this section, a summary of important issues and requirements concerning transaction systems are presented, including a discussion of how tdbm addresses them. Although we discuss transactions in the general context of a multi-threaded application, transactions can be employed to advantage in a single-threaded environment. A discussion of important aspects of the environment in which tdbm was to be used follows. An overview of recovery techniques, nested transactions, and design considerations of the external hashing component are then given. The section continues with a description of volatile and persistent databases, the Threads lightweight process kernel, and support for distributed operations.

Why atomic transactions?

An atomic transaction is a sequence of operations that are performed as a unit. The collection of operations within the scope of the transaction is executed indivisibly and in isolation from any concurrent actions outside of the transaction. The concept of indivisibility is illustrated in Figure 1. If a process executing Transaction1 explicitly aborts the transaction (e.g., because BalanceA is found to be less than $10) or the process crashes before the end of Transaction1 is reached, then neither balance should be changed. Furthermore, if two or more processes execute Transaction1 concurrently, the results should be the same as some sequence of non-concurrent executions of the transaction. This characteristic is called serializability.

```
Begin Transaction1
    Read      BalanceA
    Subtract  $10 from BalanceA
    Write     BalanceA
    Read      BalanceB
    Add       $10 to BalanceB
    Write     BalanceB
End Transaction1
```

Figure 1: A Simple Transaction

Transactions provide implicit concurrency control, freeing the programmer from the need to explicitly manage locks on objects. Lock management typically involves operations such as creating a lock, obtaining a read or write lock for an object, upgrading a read lock to a write lock, releasing a lock, and detecting and resolving deadlock.

Because transactions provide atomicity, they also simplify exception handling for the programmer since an explicit abort "undoes" a partially completed request that may involve many objects and a considerable amount of intermediate state. Transactions provide a simple and easy to use facility to create fault-tolerant applications.

The Transaction Paradigm

To achieve indivisibility, a transaction must have four properties which together form the "ACID principle" [7]: All-or-none atomic behaviour, Consistency, Isolation, and Durability. These properties are defined as follows:

- Atomicity. If a transaction successfully commits, all actions within the transaction are reflected in the database, otherwise the transaction does not modify the database at all as far as the application is concerned.
- Consistency. The consistency of the database is preserved whether the transaction commits or aborts. A database is consistent if and only if it represents the results of successful transactions.
- Isolation. The intermediate states of data manipulated by a transaction are not visible outside of the transaction. This prevents other processes from reading and acting on these intermediate results.
- Durability. Once a transaction commits, its results will survive any subsequent failures.

As part of the mechanism to attain these characteristics, the transaction system must have a recovery component that is executed when the database is opened. Recovery involves "undoing" any intermediate results applied to the database by an incomplete transaction and reexecuting any completed transactions whose results may not be fully reflected in the database.

Environmental Issues

The primary motivation behind the development of tdbm was to replace our EAN X.500 directory service's original dbm work-alike database with a more powerful and efficient one. The nature of database operations for the X.500 directory service (and directory services in general) is that there are relatively few updates compared to lookups; some of the design decisions were made in light of this observation.

An examination of the original database showed that keys were relatively short and that there was a mixture of small objects and larger objects. Table 1 shows some statistics gathered from an X.500 database consisting of about 5,900 entries and 36,249 items. Because of the way key structures were flattened into buffers, they all ended up being 20 bytes in length. These figures helped to guide the design of tdbm.

Type	Min	Max	Mean	Size
Keys	20	20	20.0	724,980
Values	38	23,122	181.4	6,577,713

Table 1: X.500 Database Item Sizes (bytes)

During the performance evaluation of the dbm work-alike database it was found that when large page sizes were used, there could be many items in a page (often several hundred) and a significant amount of time was spent searching sequentially within a page for a particular key. This information suggested a different page format.

The fact that the database is used by the object store within the directory service also influenced the design. The object store uses encoding functions to flatten and compress a data structure into a contiguous buffer before calling tdbm to store it. When fetching an item from the database, the inverse operation is performed by the object store to restore the original data structure. Because encoding and decoding always result in a new copy, tdbm gives the caller a pointer to the item. A side effect of this, however, is that the item may not be properly aligned with respect to the requirements of the processor architecture because the item's location within the buffer has been shifted during space compaction[2]. Likewise, a user storing a binary integer could encounter this problem. To solve this alignment problem, the caller can specify alignment requirements for the value so that it can be maintained within tdbm's buffer.

The directory service required only two database files and there was no need for transactions across the two databases. This was taken advantage of to simplify the implementation of recovery

Recovery Techniques

In the interest of brevity, only the major approaches to recovery will be outlined. The reader interested in more detail is referred to [7].

A recovery technique must write log information to persistent storage (e.g., disk) either so that is later possible to remove the results of incomplete transactions applied to the database (UNDO) or to apply the results of complete transactions that are not reflected in the database when it restarts after a crash (REDO). If the log contains the physical representation of objects, it is referred to as *physical logging* ; if higher-level objects are recorded then it is called *logical logging.*

If physical logging is done at the page (disk block) level, whenever any part of a physical page is modified the entire page must be written to the log. If recovery is based on the UNDO operation, the old page must be saved; if the REDO operation is used, instead of modifying the old page, the new page image is logged. It is possible to reduce the size of the log by only recording the differences between the initial and final page images. At the logical level, for example, it is possible to record the operations and their parameters in the log so that a REDO of the user's request can be executed later, if necessary.

[2]The starting address for dynamically allocated memory is typically chosen so that any data type stored there will be properly aligned. Therefore, making a copy solves the alignment problem.

When UNDO is used, log information must be written before the database is modified (called *write-ahead logging*), while when REDO is used, log information must be written before the end of transaction is acknowledged.

With each of these approaches there is a tradeoff between normal processing overhead, recovery processing cost, and implementation complexity. The degree of assistance provided by the file system is also an important factor.

Nested Transactions

If a transaction is permitted to have one or more subtransactions associated with it, a hierarchical grouping structure called nested transactions [13] result. When a child transaction commits, its state is passed up to its parent; only when a top-level transaction commits can the changes be made durable.

Nested transactions fail independently of each other; subtransactions may abort without causing other subtransactions or the entire transaction to abort. They also form a convenient unit for parallelism, with each child transaction mapping on to a thread of control. Nested transactions provide synchronization among the subtransactions, making it easy to compose new transactions from existing transactions without introducing data inconsistency arising from concurrency.

Extensible Hashing

A considerable amount of work has been done on extensible hashing schemes [18,22]. In these algorithms, a database or hash file is composed of some number of (usually) equal-length pages, with each page holding zero or more items. Most of these schemes aim to retrieve the page holding the item of interest in one or two disk operations as the hash file grows and shrinks. The goal is to maintain this performance without having to do a costly rehashing of all of the items as the size of the hash file changes. While the various approaches vary in their complexity, space overhead, and ease of implementation, they all tend to depend on a secondary data structure, such as a directory or index, to assist in locating the page containing the item. When the occupancy or load factor of a page falls outside its allowed range, a reorganization takes place; e.g., an overfilled page will be split into two partially filled pages and a record of the page split is made in the directory structure. Variations on this theme include Extendible Hashing [6,12], Dynamic Hashing [8], Linear (Virtual) Hashing [9,10,11,19], and Thompson's dbm method [24].

In some algorithms, relatively large directory structures may result. Schemes that require special page overflow handling (e.g., via bucket chaining) have additional complexity for these special cases. In the context of transactions, schemes that access many pages have the unfortunate consequence that these pages will need to be read or write locked. Apart from the extra I/O overhead of reading and writing these pages, there might be substantial concurrency control overhead.

In any of these schemes, some additional mechanism is required to handle keys or values that are too large to fit in a page. One way to handle the problem is to put a pointer to the item in the location where the item should go. The pointer could simply be the name of a separate file containing the item, as was done in our previous `dbm` work-alike. While this is relatively easy to implement, it suffers comparatively high overhead in creating, opening, closing, and removing the separate file. This can be particularly inefficient if the large item is frequently updated. Another approach is to effectively implement a simple file system so that pages can be dynamically allocated and released within the hash file. The page number generated by the extensible hashing scheme is then treated as a logical page number which is mapped to a physical page number within the hash file. Physical pages need not all be the same size. The Berkeley Hash Package (`bdbm`) [19] uses an algorithm called buddy-in-waiting to support multiple physical pages per logical page.

After surveying the literature and experimenting with several hashing schemes, it was decided to reimplement Thompson's method because its performance was known to be good, we have had considerable experience with it, and it also appears to behave reasonably well with respect to its page access patterns. We elected to implement a simple bitmap-based dynamic page allocation mechanism to manage operations on contiguous page ranges to handle the problem of indirect items. It was also decided to maintain the database within a single UNIX file to keep the design cleaner.

Volatile and Persistent Databases

In many applications, such as the directory service, it is necessary to maintain a database of persistent objects; i.e., objects written to non-volatile storage so that they can exist independently of the process that created them and survive a system crash. On the other hand, there are applications that want these objects to always be in memory, perhaps for performance reasons, or because it is unnecessary or impractical to save the objects outside of a process' volatile memory. Because of this, transactions on volatile databases should not require any disk accesses. Some applications might like the choice between volatile and persistent databases to be made at runtime; having to use a separate package such as `hsearch` [2] for volatile databases is not a viable option.

The desire to avoid a separate set of functions for each of the two database storage modes called for the notions of volatile and persistent databases to be integrated. As a consequence, the most straightforward design seemed to have all operations occur in memory up to commit time, when secondary storage became involved in the case of persistent databases. Designs involving writing intermediate states to disk were ruled out. Also, rather than implementing complicated secondary storage management functionality as for differential files, it was decided to let UNIX's virtual memory system deal with memory management.

Note that because transactions on volatile databases are not durable, no recovery component is needed for them outside of handling aborts.

Threads

Threads [15] is a lightweight process kernel that resides within a single host operating system user process. It provides fast thread creation, a shared address space for all threads, non-preemptive scheduling, and efficient IPC (using blocking Send/Receive/Reply) and context switching between threads. Of considerable significance to `tdbm`, the implementation of locks, semaphores, and shared memory does not require any intervention by the UNIX kernel since there is a single address space for all threads.

Portability is facilitated through the sub-kernel technique because instead of making system calls to the host kernel directly, applications must call the sub-kernel's versions. As a result, there is little application code that directly relies on the operating system. Threads has been ported to several different flavours of UNIX as well as several different machine architectures.

Support for Distributed Databases

One of the original design goals called for databases capable of being used with an object store that supports distributed operations and replication. The distributed object store is responsible for interprocess communication and execution of an atomic commit protocol (such as the two phase commit protocol [3]), but the underlying databases must provide some assistance.

Consider the case shown in Figure 2 where a distributed object store updates items in two or more databases within an object store transaction, each database running in a different server process, possibly on different hosts. One of the cooperating object store servers is distinguished as the transaction coordinator. After the last update in the transaction has been executed, the coordinator wants the transaction to be committed at all databases or at none. In the first commit phase, the coordinator requests all databases to ''precommit'' (prepare to commit) their part of the transaction and report the outcome. After a successful precommit, each database guarantees that a subsequent commit (the only operation allowed on the transaction beside abort) will succeed. If one or more databases fail to precommit, the transaction is aborted. If the coordinator proceeds to the second phase, all databases will be asked to commit their transaction.

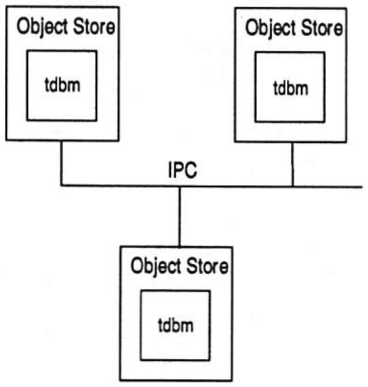


Figure 2: A Distributed Object Store

When the object store restarts, it needs a way of determining whether there was a distributed transaction in progress, and if so, which databases were involved and which phase the transaction was in. If a database crashes after its transaction is precommitted but before the second phase completes, the object store must determine whether to commit or abort the transaction when it restarts. This requires that the object store keep some state information about a transaction until it knows that all databases have committed or aborted. Also, as part of recovery, a database may need to contact the coordinator to determine the outcome of the transaction.

3. Implementation

The `tdbm` library is implemented as three independent layers: the item layer, page layer, and transaction layer. It would be relatively straightforward to replace a layer or have multiple versions of a layer. The `tdbm` library consists of approximately 6,500 lines of C source code, including header files and comments.

Although for our purposes it was not necessary to maintain compatibility with `ndbm`, the interfaces are quite similar. The major differences are that most `tdbm` functions require database and transaction identifying parameters and most functions return a result code.

The `tdbm` library can currently be compiled to run in ''non-concurrent'' mode (i.e., without any locking) and in multi-threaded mode under the Threads lightweight process kernel. In either case, `tdbm` runs as a single user-level UNIX process. When configured for concurrency, `tdbm` uses Threads' semaphores and lock manager, so no extra UNIX system calls are required.

The Item Layer

The item layer deals with the layout of key/value pairs in a page. There are two kinds of pages. The first kind is similar to that used by `dbm` and contains a directory for the items stored in the page and zero or more items. All of these pages are the same length. The second kind, indirect pages, are a variable number of physical pages long and simply contain data values that are too large to fit in any normal page.

In addition to its simplicity, the original `dbm` page format has the advantage that the contents of a page are packed so that there is no fragmentation between items. Reducing the number of pages helps to keep the database small and makes iterating through all items in the database more efficient. In the context of transactions, this is important because it reduces main memory requirements since many pages might be held in memory during the life of a transaction.

To help lower the overhead of searching for a key within a page, the `tdbm` page format was designed so that a binary search could be used to locate an item. This data structure permitted the same efficient item packing within a page at the expense of maintaining additional directory information. Keeping the directory ordered slows adding and deleting entries somewhat, but significantly reduces search times when there are many items per page.

Each normal page consists of a variable length directory and some number of items (Figure 3). The directory is a vector of unsigned short integers, beginning with a count of the number of entries in the page and `indexlast`, the index for the directory entry that points to the innermost item (i.e., the one bordering on the free space). Each directory entry consists of a tuple: `(keyoffset, keylength, dataoffset, indexnext)`. These directory entries are kept sorted by their key, in ascending `keylength` order and lexicographically for equal keylengths. The first directory entry is for the ''smallest'' key and the last is for the ''largest''. The `indexnext` fields form a circular list and keep track of the relative positions of the items; that is, `indexlast`'s `indexnext` is the index of the directory entry for the item closest to the end of the page. The `indexnext` of that directory entry is for the item second from the end of the page, and so on. If the value is stored indirectly, the value field in the current page indicates the physical page number where the real content is and the real size of the content.

Each entry has a `flag` byte that indicates the alignment requirement for the value and whether the value is stored in the current page or indirectly. The flag byte consists of four fields. The alignment constraint is specified by the user at the time the value is stored: no constraint, even address, and addresses divisible by 4 or 8. For example, a character string probably would not require special alignment but an integer might require an address divisible by 4 so that after fetching the value it can be accessed directly from the page buffer without copying. Alignment requirements for the key, as specified by the user, are currently stored but not enforced by the system. The system sets the `Indirect` bit if the value is too big to fit on a page (the key remains on the page).

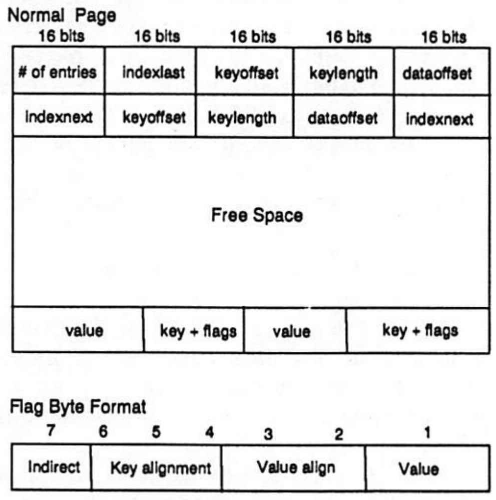


Figure 3: Page Layout

This organization provides a good tradeoff between lookup speed and space utilization, although if there are a great many very small items the directory space required per item could be unacceptable. Both the original dbm organization and the new one can spend a considerable amount of time compacting or coalescing page contents after a deletion to eliminate fragmentation. This is particularly evident in page splitting, since on average one half of the entries are deleted and moved to a new page. While this is acceptable when lookups are more frequent than updates, it may not be in the reverse situation. One solution is to allow alternative page directory formats, either per database or per page. The user could select a data structure to optimize for space or time or the system could automatically convert formats based on the space utilization.

The Page Layer

The page layer deals with management of logical pages, allocation of physical pages, page caching, and the mappings from keys to logical pages and logical to physical pages. It is not concerned with the page contents.

At the time a database is created, the user can specify the physical page size and the allocation unit size. The page size must be a power of 2 and should take into account the best I/O block sizes for the filesystem and internal fragmentation of indirect items. An entry that won't fit into a page is stored outside the normal page space and suffers an extra disk read and, likely, internal fragmentation. This internal fragmentation is ameliorated somewhat by having an allocation unit size in addition to the page size. The allocation unit size is the size of each page in the file as far as page allocation is concerned.

In addition to configuration constants like the pagesize, the header of the database contains three tables: the splitmap, physmap, and freemap. The splitmap is equivalent to the `.dir` file of a `dbm` -style database. It keeps track of how many times each page has split. The physmap maps a logical page number to a starting physical page number. The maximum size of these tables, in pages, is determined at compile-time. The maximum number of logical pages is (`NPHYSMAP_PAGES` * *pagesize*) / *sizeof(u_int)*, where `NPHYSMAP_PAGES` is a compile-time constant.

The freemap is a bitmap representation of physical page allocation. Each bit in the freemap represents *(pagesize / allocunits)* bytes. For example, if *pagesize* is 8192 and *allocunits* is 8, then disk space allocation is in blocks of 1024 bytes and 8 contiguous blocks are needed to allocate one page. As *allocunits* gets larger, external fragmentation tends to grow but internal fragmentation in indirectly stored data tends to decrease. With a *pagesize* of 8192 bytes, one freemap page contains (8192 bytes/page * 8 bits/byte) == 65536 bits/freemap page, which can map (65536 pages * pagesize bytes/page) == 536Mb. Physical page 0 is the file header and pages `INITIAL_FREEMAP_PAGE` through (`INITIAL_FREEMAP_PAGE + NFREEMAP_PAGES - 1`) are preallocated for the freemap. Pages after this last page are dynamically allocated and page `FIRST_AVAIL_PAGE` will be the first physical page represented in the freemap. While this approach leaves several holes at the beginning of the file (making it look much bigger than it really is), it simplifies the problem of having the freemap allocate new space for itself.

Here are the values currently being used, requiring a minimum page size of 2048 bytes to accomodate the header:

```
#define NSPLITMAP_PAGES          100
#define NPHYSMAP_PAGES           400
#define INITIAL_FREEMAP_PAGE     1
#define NFREEMAP_PAGES           10
#define FIRST_AVAIL_PAGE         \
    (INITIAL_FREEMAP_PAGE + NFREEMAP_PAGES)
```

The hash function that maps the user's key string to an integer is part of the page layer. A good hashing function is critical to the performance of any extensible hashing package. Nine functions were evaluated by having each hash a list of 84,165 strings made up of English words and symbol table entries and counting the number of collisions. One of the best, the hash function from sdbm [25] (also used in bdbm), was chosen for tdbm because it did not generate a single collision.

The Transaction Layer

The transaction layer provides nested transactions over logical pages. The transaction layer is responsible for locating the appropriate version of a page for a transaction, concurrency control, and commit and recovery processing.

Every page that is read from the database is cached as a top-level or base copy [3]. All transactions that read the page share this base copy. If a subtransaction updates or creates a page, it retains a private copy of the page that may be later accessed by it and its subtransactions. The correct instance of a page for a particular subtransaction is quickly located by associating a simple hash table with each transaction identifier. The search process involves examining the transaction's hash table, proceeding up the hierarchy through its superiors' hash tables, and finally reading the database, if necessary, until the page is found. As transactions commit, their state is merged with that of their parent; when a top-level transaction commits, the new pages are propagated back to the hash file.

Apart from reading base pages from the database during transactions, writing pages to the transaction file while preparing to commit, and copying pages from the transaction file to the database during the commit, no other file I/O is performed. When dynamically allocated pages are no longer required, they are freed for general use by the application. Keeping all accessed pages in virtual memory eliminates any I/O from a temporary file and makes the integration of persistent and volatile databases seamless. It was felt that performance would be better if a temporary file could be avoided and that, in our environment, long-running transactions are unlikely. A shortcoming of this approach is that the number of pages a transaction can access is limited by the constraints of UNIX's virtual memory subsystem; a transaction will fail if it requires more memory than is available. Bdbm, on the other hand, may use a temporary file in conjunction with a volatile database if its cache size is exceeded. For best performance, the bdbm user has to specify a sufficiently large cache size at the time the database is opened. This memory is reserved until the database is closed.

Currently, the programmer is responsible for synchronizing threads executing subtransactions with that of the parent transaction; a parent transaction cannot commit or abort until all of its subtransactions have terminated. Automatic blocking in these cases is a possible extension to the package.

[3]Currently, this copy is kept in memory only as long as some transaction needs it.

Concurrency Control

Concurrency control is only performed when tdbm runs under Threads; it is simply not configured into the system otherwise. When concurrency control is available, a particular database can be opened multiple times by different threads and there can be many concurrent tdbm transactions on the database.

Threads provides a lock manager that allocates, obtains, and releases a lock on behalf of a client thread. The tdbm library uses Threads semaphores to protect critical sections. As a tradeoff between overhead and the level of concurrency, concurrency control is performed at the page level rather than at the hash file level (as our work-alike did) or the item level. Before a page can be read, tdbm obtains a read lock for the page. Tdbm must obtain a write lock for a page, perhaps by upgrading a read lock that the transaction already holds, before a page can be written. In keeping with strict two phase locking [13], locks are not released until the top-level transaction commits or an abort occurs. When a subtransaction commits, the parent transaction inherits any locks.

A good deal of the complexity of the transaction layer is due to the lock management protocol required by nested transactions and sharing unmodified pages. Only a limited degree of deadlock detection is currently implemented.

Commit Processing and Recovery

Commit processing of a top-level tdbm transaction is done by creating a transaction file (also called an intention list [23]) that represents the actions that must be executed to update the database. This approach is known as *after-image physical logging* [7]. The transaction file is stored in the same directory as the database file.

The transaction file contains some header information followed by a variable number of fixed-length shadow pages that represent the new contents of physical pages in the database. When the transaction file has been written and secured to disk using the fsync() system call, a chmod() is done to it to atomically indicate that the transaction has been committed. At this point, the new pages in the transaction file can overwrite the old pages in the database file. Upon successful completion of top-level commit processing, the results of the transaction have been applied to the database and the transaction file can be removed. This technique is similar to the idea of differential files [21], although tdbm operations never access the transaction file.

Recovery is automatically initiated when tdbm is started so that incomplete transaction files (those without a file mode indicating they've been committed) can be removed and the contents of completed transaction files can be applied (or reapplied) to the database. Note that this recovery procedure is idempotent; if the system crashes during the overwriting process, recovery can be retried until successful.

This approach to recovery was taken primarily because of its simplicity. For example, it is not necessary to make a copy of the old data. Also, it was used by the package `tdbm` replaced and known to work well in practice. Physical logging was chosen because it was straightforward to make commit processing atomic and idempotent; it was felt that it would be more difficult to implement and debug logical logging of operations. Alternate approaches require maintaining log information, perhaps in terms of `tdbm` commands rather than disk pages (logical logging), so that changes can be undone or reapplied. A disadvantage to this approach is that a number of small changes to many pages may result in a large transaction file, but this can be mitigated by careful choice of page size and was not deemed to be as significant as benefits arising from the scheme's simplicity.

Since modified pages are not directly written to the database file, no log information needs to be maintained. On the other hand, modified pages must be held in memory until commit time, effectively caching all accessed pages. For transactions that do not involve a huge amount of data, this is not a significant penalty in terms of memory requirements and should normally result in improved performance. Also, this greatly simplified implementation of volatile (non-persistent) databases since virtually all of the same code can be shared with that necessary for persistent databases. The same storage and retrieval algorithms are used whether the database is volatile or persistent.

Several extensions to this basic scheme were necessary to support two phase commit. A precommit operation was added for top-level transactions. It is called to precommit any local updates within the transaction and to track the state of a transaction that has successfully completed the first phase. In either case, the caller can associate arbitrary data with the precommit. If, at the time the database is opened, transactions are found that terminated after phase one but before phase two completed, the application receives an appropriate return code. It may then invoke the recovery operation to obtain the data associated with the transaction and subsequently complete the transaction. In this way, the application can update the transaction's global state. The database is unavailable for normal operation while recovery is going on. The transaction can be committed (and the transaction file removed) when all databases have agreed on the outcome.

4. Evaluation

In this section, the performance of `tdbm` is compared to that of the most widely-used extensible hashing library under UNIX, `ndbm`, and the package expected to be its replacement, `bdbm`. All experiments were performed on a Sun Sparcstation 1 running SunOS 4.1.1 with 24Mb of memory and a CDC Wren IV disk drive.

In the first experiment, the performance of the three libraries was measured for creating and retrieving data from persistent databases, varying page sizes and the amount of data being stored. The second experiment involved repeating the first series for `tdbm` and `bdbm` using volatile databases. In all cases, consecutive integers (as ASCII strings) were used as keys. Three different sets of data values were used; they are characterized in Table 2. The first is the list of words in `/usr/dict/words`, the second is the word list referred to in Section 3, and the third is a set of 200 RFC documents. All times reported are the sum of the times spent in user mode and system mode. The error in these measurements is approximately ±5%. `Tdbm` was tested outside of threads, so there is no concurrency control cost associated with its measurements. Each `tdbm` test run involves a single transaction.

To reduce the number of experimental factors being varied, the `bdbm` fill factor parameter[4] was set at 128 for all `bdbm` runs since that value tended to result in the best performance for all page sizes [19]. Also, `bdbm` was run with both the default cache size of 65,536 bytes and a size of 4Mb. The latter configuration was the smallest size that virtually eliminated `bdbm`'s need to access a temporary file. Since `ndbm`'s page size can't be determined at run time, several versions were compiled.

[4]The fill factor indicates a desired density within the hash table and approximates the number of keys allowed to accumulate in any one bucket.

Name	Number of Values	Value Sizes (bytes) Minimum	Maximum	Mean	Total
`/usr/dict/words`	25,144	1	22	7.2	206,672
`wordlist`	84,165	1	40	8.9	836,663
`rfc`	200	143	799,768	78,990.6	15,877,102

Table 2: Data Value Sets

Persistent Databases

The first experiment examined creating and reading persistent databases. Figure 4 shows the time to create a persistent database using the contents of `/usr/dict/words` as data values. The results of fetching all data items by using the data keys sequentially are shown in Figure 5 and the results of having the databases iterate through each key are shown in Figure 6.

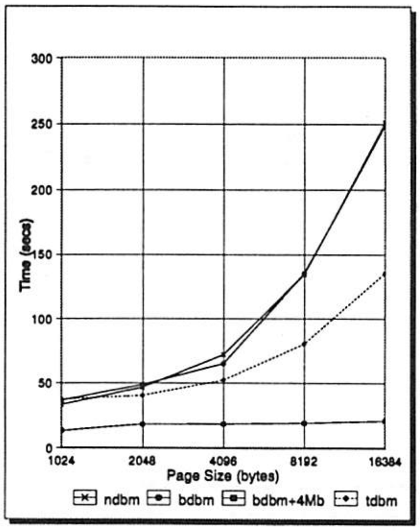


Figure 4: Creating Persistent Databases (`/usr/dict/words`)

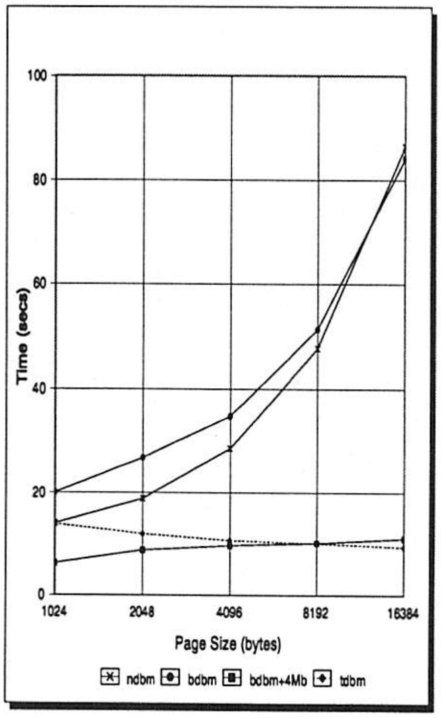


Figure 5: Reading Persistent Databases (`/usr/dict/words`)

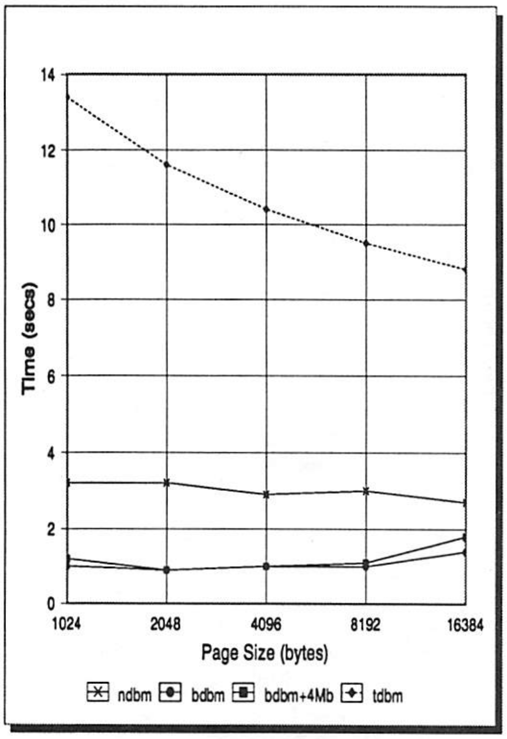


Figure 6: Iterating Through Persistent Databases (`/usr/dict/words`)

`Tdbm` performs well compared to both `ndbm` and `bdbm` in the creation and reading tests. A trial where the data keys were shuffled for the reading test did not yield significantly different results from their sequential use. Because `tdbm` keeps all pages accessed during a transaction in memory, it performs relatively poorly on the iterative key retrieval test. In fact, this operation is almost functionally identical to that used by the sequential reading test so this test was not repeated in the other configurations.

The next set of runs (Figures 7 and 8) repeats the previous set with the larger set of data values in `wordlist`. Here, `tdbm` performs substantially better than both `ndbm` and `bdbm` with the small cache. The last data point in Figure 8 for `bdbm` with 4Mb cache probably is indicative of the performance hit taken when `bdbm` switches to its temporary file after its cache fills.

The last run in the series (Figure 9) shows that `tdbm` performs well in conjunction with large data values. Since `ndbm` cannot load data values larger than its page size, it was excluded from the `rfc` data value runs.

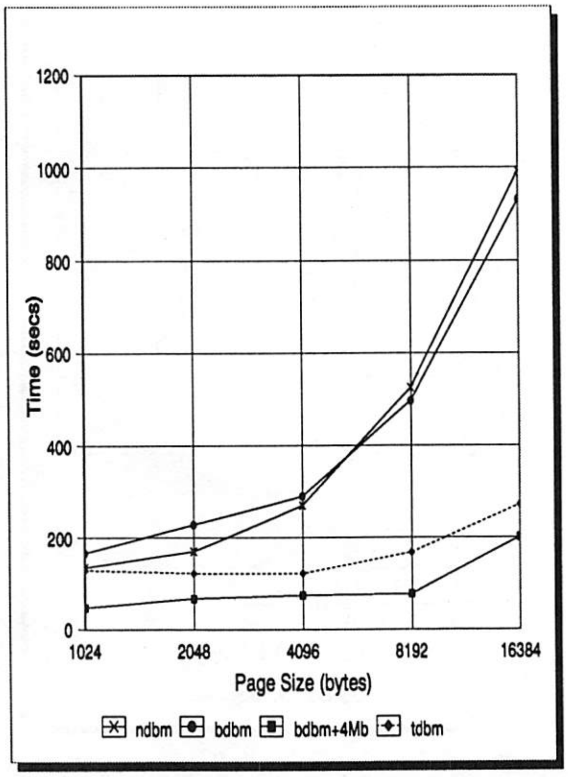


Figure 7: Creating Persistent Databases (`wordlist`)

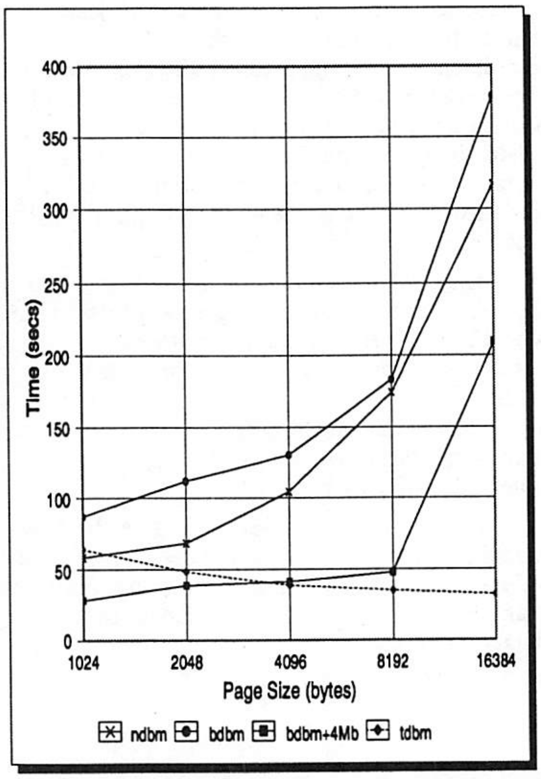


Figure 8: Reading Persistent Databases (`wordlist`)

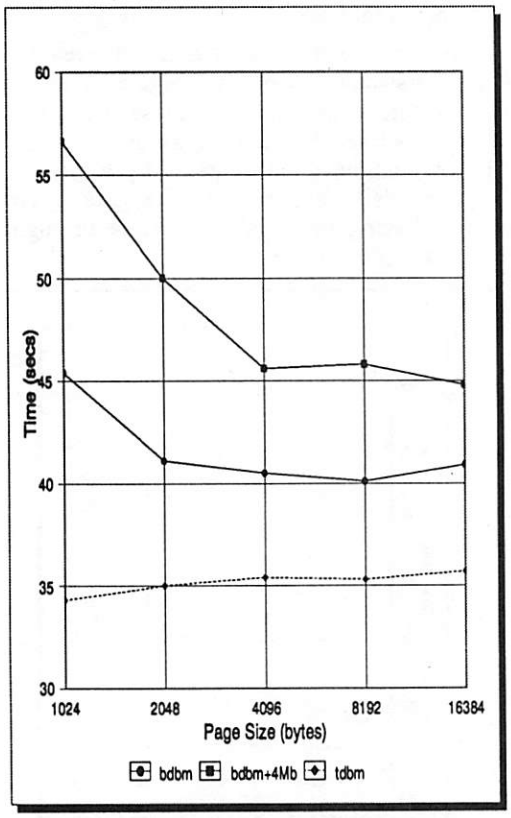


Figure 9: Creating Persistent Databases (`rfc`)

When creating a new database, all libraries tend to prefer a small page size. The exception is the `rfc` test where bdbm does better with a larger page size. While ndbm and bdbm also prefer a small page size for reading, tdbm does better as the page size increases. These observations are also true for volatile databases, discussed in the next section.

Volatile Databases

The second experiment examined the performance of bdbm and tdbm with volatile databases (ndbm doesn't support volatile databases). For both the `/usr/dict/words` and `wordlist` data value sets, tdbm performs about the same as for a persistent database. This indicates that for a modestly-sized update, tdbm's commit processing is not expensive. Note that with the small cache, bdbm's performance is slower than for a persistent database. Figure 14 shows that for very large updates, tdbm outperforms bdbm. Tdbm's commit cost for very large updates can be seen by comparing its performance in Figures 9 and 14.

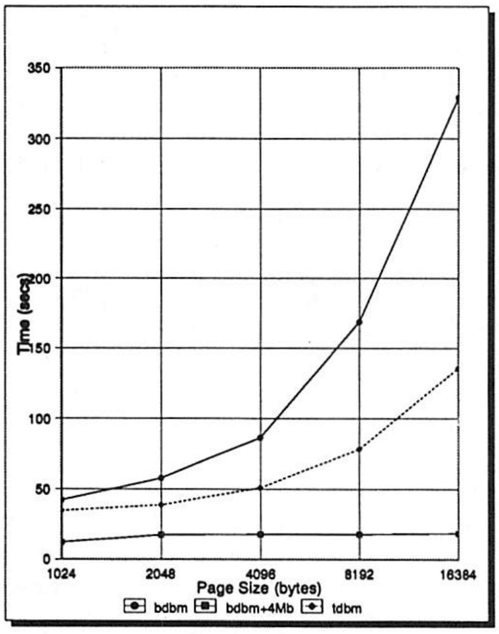


Figure 10: Creating Volatile Databases (`/usr/dict/words`)

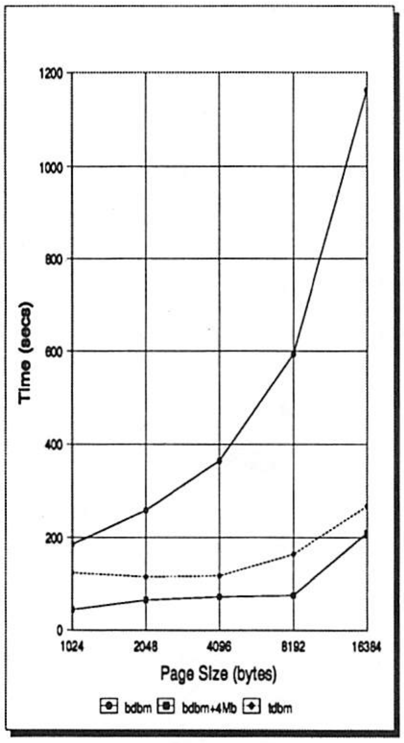


Figure 12: Creating Volatile Databases (`wordlist`)

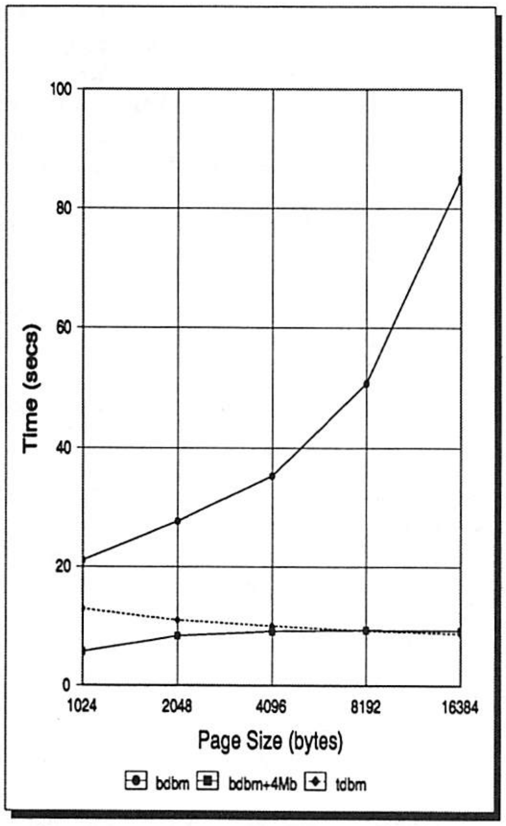


Figure 11: Reading Volatile Databases (`/usr/dict/words`)

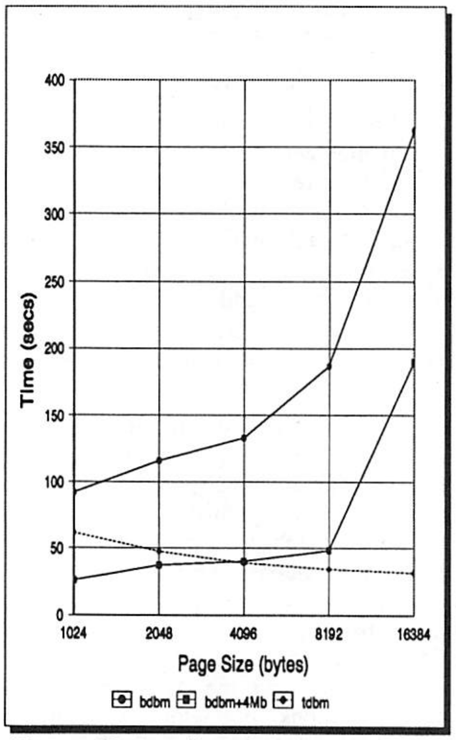


Figure 13: Reading Volatile Databases (`wordlist`)

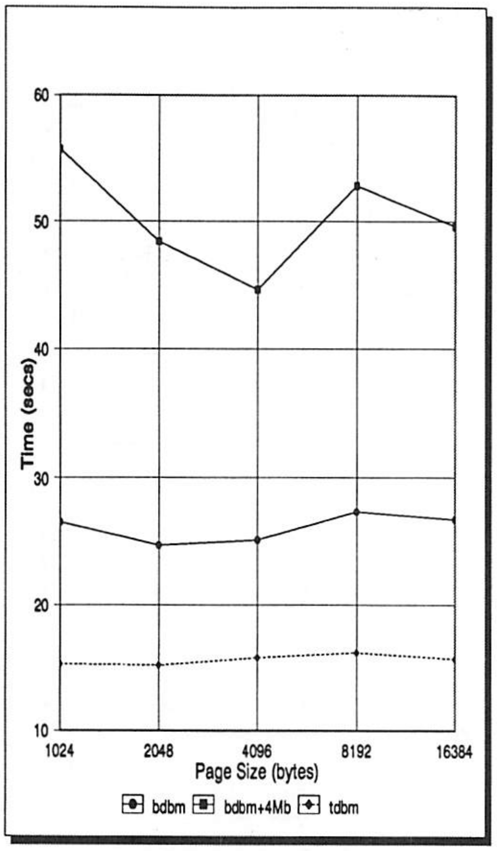


Figure 14: Creating Volatile Databases (rfc)

5. Conclusions

The tdbm library has been incorporated into the experimental version of the object store used by the EAN X.500 directory service. It has successfully met its design goals as a component of the directory service and, although there is room for improvement, we are satisfied with it. It performs well in comparison with ndbm and bdbm while providing important new features, such as nested atomic transactions, fault tolerance, and multi-threaded operation.

There is certainly much more that could be done to improve tdbm. This might include features such as base page caching (pages that are no longer referenced are currently flushed at top-level commit time), multiple database transactions, multi-volume databases (one database spread over multiple file systems), user selectable page formats and user maintained page formats, and statistics gathering. There is currently no page unsplitting, which would make iterating through the database more efficient after many deletions, but which probably wouldn't reduce the size of the database file under UNIX. Some simpler features may eventually be added, such as arbitrary size keys, user-specifiable hash functions, and deadlock detection, however we currently have no need for them.

Performance evaluation of tdbm under Threads, using both real and apparent concurrency, is planned. It might also be interesting to configure tdbm to run under SunOS's or Mach's threads. Separating the atomic transaction mechanism into a separate UNIX process could also be examined.

6. Availability

The library will be made available for anonymous ftp from `cs.ubc.ca` (137.82.8.5) as `pub/local/src/tdbm.tar.Z`.

7. Bibliography

[1] AT&T. **dbm(3X)**, *UNIX Programmer's Manual*, Seventh Edition, Volume 1, Bell Laboratories, Jan. 1979.

[2] AT&T. **hsearch(BA_LIB)**, *UNIX System User's Manual*, System V.3, 1985, pp. 506-508.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. ''Concurrency Control and Recovery in Database Systems'', Addison-Wesley, 1987.

[4] CCITT. ''Recommendation X.500: The Directory – Overview of Concepts, Models and Services'', Dec. 1988.

[5] Computer Systems Research Group, Computer Science Division, EECS. **ndbm(3)**, *4.3BSD UNIX Programmer's Reference Manual (PRM)*, University of California, Berkeley, Apr. 1986.

[6] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. ''Extendible Hashing – A Fast Access Method for Dynamic Files'', *ACM Trans. on Database Systems*, Vol. 4, No. 3, (Sept. 1979), pp. 315-344.

[7] T. Haerder and A. Reuter. ''Principles of Transaction-Oriented Database Recovery'', *Computing Surveys*, Vol. 15, No. 4, (Dec. 1983), pp. 287-317.

[8] P. Larson. ''Dynamic Hashing'', *BIT*, Vol. 18, 1978, pp. 184-201.

[9] P. Larson. ''Linear Hashing with Partial Expansions'', *Proc. 6th Int. Conf. on Very Large Data Bases*, 1980, pp. 224-232.

[10] P. Larson. ''Linear Hashing with Separators – A Dynamic Hashing Scheme Achieving One-Access Retrieval'', *ACM Trans. on Database Systems*, Vol. 13, No. 3, (Sept. 1988), pp. 366-388.

[11] W. Litwin. ''Linear Hashing: A New Tool for File and Table Addressing'', *Proc. 6th Int. Conf. on Very Large Data Bases*, 1980, pp. 212-223.

[12] D. Lomet. ''Bounded Index Exponential Hashing'', *ACM Trans. on Database Systems*, Vol. 8, No. 1, (Mar. 1983), pp. 136-165.

[13] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[14] P. Nelson. ''gdbm source distribution and README file'', Version 1.5, Feb. 1991.

[15] G. Neufeld, M. Goldberg, and B. Brachman. ''The UBC OSI Distributed Application Programming Environment – User Manual'', Technical Report 90-37, Department of Computer Science, University of British Columbia, Jan. 1991.

[16] G. Neufeld, B. Brachman, M. Goldberg, and D. Stickings. ''The EAN X.500 Directory Service'', Technical Report 91-29, Dept. of Computer Science, University of British Columbia, Nov. 1991.

[17] G. Neufeld and B. Brachman. ''The EAN Object Store'', in preparation, 1992.

[18] B. Salzberg. *File Structures: An Analytic Approach*, Prentice-Hall, 1988.

[19] M. Seltzer and O. Yigit. ''A New Hashing Package for UNIX'', *Proc. of the Winter Usenix Conf.*, Usenix Association, Jan. 1991, pp. 173-184.

[20] M. Seltzer and M. Olson. ''LIBTP: Portable, Modular Transactions for UNIX'', *Proc. of the Winter Usenix Conf.*, Usenix Association, Jan. 1992, pp. 5-25.

[21] D. Severance and G. Lohman. ''Differential Files: Their Application to the Maintenance of Large Databases'', *ACM Trans. on Database Systems*, Vol. 1, No. 3, (Sept. 1976), pp. 256-267.

[22] P. Smith and G. Barnes. *Files & Databases: An Introduction*, Addison-Wesley, 1987.

[23] H. Sturgis, J. Mitchell, and J. Israel. ''Issues in the Design and Use of a Distributed File System'', *Operating Systems Review*, Vol. 14, No. 3, (July 1980), pp. 55-69.

[24] C. Torek. ''Re: dbm.a and ndbm.a archives'', *USENET newsgroup comp.unix*, Apr. 17, 1989. This article discusses the structure of dbm databases and the algorithms used by the library.

[25] O. Yigit. ''sdbm – Substitute DBM or Berkeley ndbm for Every UN*X Made Simple'', sdbm source distribution, Dec. 1990.

Author Information

Gerald W. Neufeld is an assistant professor in the Department of Computer Science at the University of British Columbia, Vancouver, B.C. V6T 1Z2. Neufeld is the director of the Open Distributed Systems Group. His interests include computer communications, distributed applications, and distributed operating systems. He is currently working on the Raven project; an object-oriented distributed system. Neufeld received a Ph.D. in computer science from the University of Waterloo in 1987. He received a B.Sc. (Honours) and M.Sc. from the University of Manitoba. Reach him electronically at neufeld@cs.ubc.ca.

Barry Brachman is a research associate and lecturer in the Department of Computer Science at the University of British Columbia. His research interests include distributed systems, computer communications, and operating systems. He has recently been involved in the design and implementation of the EAN X.500 Directory Service and distributed databases and object stores. Dr. Brachman received the B.Sc. Honours degree from the University of Regina in 1981, and the M.Sc. and Ph.D. degrees from the University of British Columbia in 1983 and 1989, respectively, all in Computer Science. Contact him via e-mail at brachman@cs.ubc.ca.

NAME

tdbm – dbm database functions with nested atomic transactions

SYNOPSIS

```
#include <tdbm.h>

DbmRc DbmAbort(tid)
Tid *tid;

DbmRc DbmBegin(parent, child)
Tid *parent;
Tid **child;

DbmRc DbmClose(dbm)
Dbm *dbm;

DbmRc DbmCommit(tid)
Tid *tid;

DbmRc DbmDelete(dbm, tid, key)
Dbm *dbm;
Tid *tid;
Datum key;

char *DbmErrorString(rc)
DbmRc rc;

DbmRc DbmFetch(dbm, tid, key, value)
Dbm *dbm;
Tid *tid;
Datum key;
Datum *value;

DbmRc DbmFind(dbm, tid, key, value)
Dbm *dbm;
Tid *tid;
Datum key;
Datum *value;

DbmRc DbmFirstkey(dbm, tid, key)
Dbm *dbm;
Tid *tid;
Datum *key;

DbmRc DbmNextkey(dbm, tid, key)
Dbm *dbm;
Tid *tid;
Datum *key;

DbmRc DbmOpen(pathname, type, config, dbm, recovery)
char *pathname;
DbmFileType type;
DbmConfig *config;
Dbm **dbm;
DbmRecovery **recovery;
```

```
DbmRc DbmPrecommit(tid, tidname)
Tid *tid;
DbmTidName *tidname;

DbmErrorClass DbmRcClass(rc)
DbmRc rc;

DbmRc DbmRecover(dbm, recovery, tid)
Dbm *dbm;
DbmRecovery **recovery;
Tid **tid;

DbmRc DbmStore(dbm, tid, key, value, mode)
Dbm *dbm;
Tid *tid;
Datum key;
Datum value;
DbmStoreMode mode;
```

DESCRIPTION

Tdbm is a collection of functions that implement a simple database made up of key/content pairs. While similar to the UNIX **dbm**(3) and **ndbm**(3) packages, there are a number of significant differences:

- Nested atomic transactions are supported across a single database.
- Volatile (temporary and memory resident) databases can be used.
- A database is implemented as a single file.
- Very large objects can be stored.
- In a multi-threaded environment, concurrent transactions are possible.
- In many cases, performance should be improved.

The usual **Datum** data structure has an additional component, a datum descriptor:

```
  typedef u_char EntryDesc;

  typedef struct {
    char        *dptr;
    int         dsize;
    EntryDesc   desc;
  } Datum;
```

The descriptor can be used to specify the alignment requirements of a key or value. For the value, the system preserves the requested alignment when it is read so that it can be accessed directly from the system buffer, obviating the need to copy the value into properly aligned memory. For the key, the alignment is not preserved but the specified alignment can be determined later.

The low-order two bits of the descriptor specify alignment. The following constants are defined:

ALIGN0 - No alignment required

ALIGN2 - Align on any even address

ALIGN4 - Align on any address divisible by 4

ALIGN8 - Align on any address divisible by 8

Before a database can be used, it must be opened by calling **DbmOpen()**. The **pathname** argument identifies the database to be used; it is created if necessary. If **type** is **DBM_PERSISTENT**, then the database will be a normal Unix file. The file will be exclusively locked using **flock**(2). If it is **DBM_VOLATILE**, then the database is temporary and will disappear when it is no longer referenced (or when the program terminates). A database can be opened multiple times except for one special case. For volatile databases, if **pathname** is NULL or the null string, then a unique internal name is effectively assigned to the database.

If not NULL, **config** allows the user to override system defaults for database parameters: If **recovery** is not NULL, a list of precommitted transactions is returned (see **DbmPrecommit**() and **DbmRecover**()).

```
typedef struct DbmConfig {
  int       mode;       /* Mode when creating new dbm */
  int       pagesize;   /* Size of each page (bucket) in the dbm */
  int       allocunits; /* Allocation units, w.r.t. pagesize */
} DbmConfig;

typedef struct DbmTidName {
  int       hostid;
  int       start_time;
  int       count;
} DbmTidName;

typedef struct DbmRecovery {
  char       *pathname;
  DbmTidName  tidname;
  struct DbmRecovery *next;
} DbmRecovery;
```

A default is overridden only if a configuration option is non-zero. An identifier associated with the opened database is returned.

Most operations on a database occur within the context of a transaction. A transaction is initiated by calling **DbmBegin()**. If the **parent** argument is NULL, then this is a top-level transaction, otherwise the new transaction is a subtransaction of **parent**. A new transaction identifier **tid** is returned. In the current implementation, all operations within a top-level transaction must be associated with a single database; a transaction is implicitly associated with a database based on the first I/O operation it performs.

N.B. The value (or key) returned by a function must be copied if it is going to be modified. Also, a datum's **dptr** may no longer be valid if the transaction with which it is associated aborts.

New entries are stored in the database using **DbmStore()**. The given **value** is put into the database using **key**. The **mode** argument is either **DBM_REPLACE** or **DBM_INSERT**. The former deletes an existing instance of an entry with the same key before storing the new entry while the latter insists that there be no existing entry. The **key** and **value** are copied, so they may be freed after this call.

DbmFetch() is used to retrieve an entry. **DbmFind**() locates an entry without retrieving the value. Both functions set the descriptor for the key and the value.

An entry is deleted from the database by calling **DbmDelete**( ).

A database can be traversed in (apparently) random order. **DbmFirstkey**() retrieves the key of the "first" entry in the database. **DbmNextkey**() may be called to retrieve the keys of successive entries. Both functions set the descriptor for the key. Note that these functions should be used carefully since they could end up locking the entire database while their transaction exists.

A transaction is committed by calling **DbmCommit**() or aborted by calling **DbmAbort**(). Aborting a transaction rolls back all modifications made to the database by the transaction. In either case, the transaction must not have any subtransactions.

A top-level transaction may be prepared for commitment, but not actually committed, by calling **DbmPrecommit**( ). Upon successful completion, the system guarantees that a subsequent commit will succeed (modulo media failures). After this call, the only operations allowed on the transaction are **DbmCommit**() and **DbmAbort**(). If **tidname** is not NULL, then it is initialized to the (globally unique) name of the transaction. If **DbmOpen**() returns recovery information, then one or more precommitted transactions exist. **DbmRecover**( ) should be called (until ***recovery** is NULL) to return a transaction identifier and the transaction name for each precommitted transaction. These functions are expected to be used by a two phase commit protocol.

A database is closed using **DbmClose**(). Closing a database aborts any active transactions using it. The given **dbm** identifier is invalidated, yet if the database was opened multiple times, operations on the database may continue using the other identifiers.

DbmErrorString( ) returns an error message corresponding to the given result code. **DbmRcClass**() returns an indication of the type of error that has occurred (e.g., **DBM_FATAL**).

WARNING

It is unwise for a program to update a database that it is accessing via NFS; databases should always be accessed directly on the file's server. NFS file locking is not performed. Also, byte ordering is not addressed by the current implementation.

FILES

As part of the atomic commit operation, a transaction file is created. Its name is that of the database with a "**.trans**" appended. The mode of this file has special meaning to *tdbm* and therefore should not be changed by the user.

SEE ALSO

dbm(3), **ndbm**(3).

The GNU gdbm library, Philip A. Nelson <phil@cs.wwu.edu>, Computer Science Department, Western Washington University.

The Berkeley Hash package, Margo Seltzer <margo@postgres.berkeley.edu>. See: "A New Hash Package for UNIX" by Margo Seltzer and Ozan Yigit in Winter 1991 Usenix Proceedings.

An interactive program, **tdbm**(1), is available to manipulate and inspect **tdbm** files.

DIAGNOSTICS

When an unusual error occurs, a message is likely to be printed on **stderr**.

BUGS

Because creation of a new database occurs outside of the transaction mechanism, rollback is not possible.

Multithreaded operation is currently supported only for the U.B.C. Threads kernel.

There are currently configuration-dependent limits on the length of a **key** (it must fit into a database ''page'') and the length of a **value** (depends on the size of the page allocation bit map and other factors).

Although very large, there are limits on the maximum size of a database. A database cannot span filesystems (unless the O/S provides it transparently).

A mechanism to support efficient tree locking should be added.

The space used by deleted entries is not reclaimed by the operating system but may be reused by the database. The database file may contain holes, making it appear much bigger that it really is. There is no automatic ''shrinking'' of the database. If desired, this must be done by traversing the old database, copying each entry to a new database.

The package cannot recover from media failures.

The location of the transaction file is not user configurable.

The obvious method of deleting (or adding) items while iterating through the database using **DbmFirstKey(), DbmNextKey(),** is buggy. No updates should be performed until you're done iterating.

The wish list of additional functionality is too long (but suggestions and bug reports are welcome). The following user-visible improvements currently top the list:

- Deadlock detection (eg., via timestamps).
- Performance instrumentation ''**DbmStats()**''.
- Make **DbmRc** a structure that includes **errno**.
- Arbitrary size keys.
- Multi-volume dbms (e.g., make it possible to allocate pages on a different filesystem once a limit is reached (soft or hard)).
- Multi-dbm transactions.

AUTHOR

Barry Brachman <brachman@cs.ubc.ca>
Dept. of Computer Science
University of British Columbia

VNS Retriever: Querying MEDLINE over the Internet

Kevin Brook Long, Jerry Fowler, Stan Barber – Baylor College of Medicine

ABSTRACT

Academic medical centers around the country are developing networking infrastructures and connecting to the Internet. Baylor College of Medicine is developing VNS Retriever, an architecture for comprehensive handling of our institution's database requests. The first implemented instance of this architecture is the MEDLINE Retriever, a tool to query the considerable citation database of medical literature at the National Library of Medicine. Response times that we have experienced with the MEDLINE Retriever have pleased us and impressed our user community. The system will work for small sites, but is extensible for use on large campuses as well. The MEDLINE Retriever uses the Corporation for National Research Initiatives' ABIDE gateway. The principal user interface employs Motif and the X Window System.

Introduction

The information consumer in the academic medical center environment is faced with a modern dilemma; as electronic, networked informatics resources become more common, users are presented with a disparate collection of interfaces, mostly designed independently as islands of information whose only integration with each other is the coincidence of existing on the same network. Although the presence of high-speed networks might appear to bring this electronic information nearer the user, it is instead often unreachable behind a wall of technical detail. We are attempting to develop and apply informatics architectures that more seamlessly integrate the information from certain classes of these resources into the information user's environment. Specifically, we have applied the concept of the layered architecture and the intelligent retrieval engine to facilitate interaction with networked bibliographic resources.

At Baylor College of Medicine, our participation in the National Library of Medicine's Integrated Academic Information Management System (IAIMS) program has involved exploring ways to apply advancements in computing and informatics to the problems of biomedicine. We have developed an integrated networked environment for coordinating human, computer, and informatic resources. The centerpiece of our work is the Virtual Notebook System (VNS), an electronic analogy to the traditional paper notebook [1,2,3]. The VNS is the realization of an architecture that supports the collaborative research environment. The work described here is a natural extension of this effort, a step closer to the longer-range goal of establishing intelligent, networked information clearinghouses.

Recognizing the growing demand in our biomedical research community for rapid access to remote electronic databases, both bibliographic and otherwise, we have devised VNS Retriever, an architecture to facilitate user access to a variety of local and remote electronic databases. The cornerstone of VNS Retriever is a central resource server that acts as an agent between the user and the NLM via asynchronous electronic message passing. At the front-end, we have implemented a user interface written using the X-Window System and the OSF Motif toolkit so that many users of a wide variety of platforms can access this new service [4,5]. As a back-end connection to the MEDLINE database, our resource server makes use of the ABIDE gateway and Knowbot Operating Environment developed by the Corporation for National Research Initiatives (CNRI) [6].

This paper discusses VNS Retriever, a layered architecture for wide-area database query and retrieval, and our first implementation of VNS Retriever, the MEDLINE Retriever. In the next section, we amplify on the motivations for this project. Thereafter, we describe the architecture of VNS Retriever, including the communications interfaces between components. Then we discuss the implementation of the MEDLINE Retriever in order to clarify the structure of the central components of VNS Retriever. We then discuss administration of the MEDLINE Retriever, and present some simple performance analysis, before discussing future work and drawing conclusions.

Motivation for VNS Retriever

The MEDLINE Retriever began as an effort to facilitate Internet access from networked workstations to the National Library of Medicine's MEDLINE facility. It has become the embodiment of an architecture to treat bibliographic information in a consistent fashion, and to assist the user in overcoming many of the technical details that often impede bibliographic research. At a more general level, VNS Retriever is an architecture that attempts to add (among other things) systemized resource characterization, automated query generation, and intelligent filtering mechanisms to the overall confines of the VNS architecture, which by contrast deal more with the integration, annotation, organization and dissemination of information.

We anticipate that VNS Retriever will be applied to a variety of resource types, including bibliographic (such as is addressed by the MEDLINE Retriever), genetic (such as the GENBANK Retriever), programmatic, textual, video, etc. The MEDLINE Retriever is our first instantiation of this architecture. The advantages that the MEDLINE Retriever brings to MEDLINE access are multiple:

- Database access software can be maintained centrally and singly for all networked users at a given site, allowing updates to search strategies, the MeSH vocabularies, access methods, etc. to be implemented immediately. This greatly simplifies the distribution of software and documentation updates.
- Users need only an X-capable display and a connection to the network to begin using VNS Retriever. It is no longer necessary for a user to procure a modem, a copy of the access software itself (such as Grateful Med [7]), a list of network access numbers, and perhaps even a personal NLM account.
- Users of hundreds of other presently unsupported computing configurations would be able to gain access to MEDLINE.
- This centrally-maintained, globally-available access to the NLM's resources could help to eliminate an institution's need to acquire a local version of MEDLINE.
- Users who travel frequently can maintain access to their queries and citations from anywhere on the network. This is consistent with a model that allows the entire network to become a support mechanism for the user, not just one machine.
- Institutional database usage patterns could be monitored centrally with the goal of understanding and providing better for the needs of the institution's research community and possibly optimizing access methods for the most used resources.
- Frequently in the academic environment, only by enriching the networked environment will users be attracted to a more distributed computing model; we feel that the MEDLINE Retriever adds value to institutionalized networking.

Architecture Of VNS Retriever

The architecture created to accomplish this task is not simply a MEDLINE-specific interface. Logically speaking, the VNS Retriever architecture has 4 layers, as depicted in Figure 1. User interfaces communicate through an application program interface with a user agent called the query manager. The query manager maintains the user's environment, storing queries and presenting their results in the manner desired by the user. The user agent, in turn, submits queries to a server agent called the resource server. The resource server selects the appropriate resources for each query, translates the query appropriately for the resource, and initiates a resource query engine to communicate with the resource. At the bottom layer are the query engines themselves, which are specialized to communicate with the individual resources.

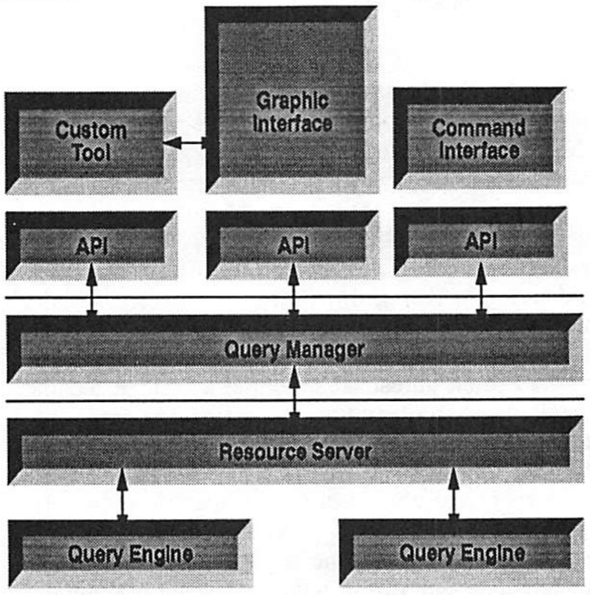


Figure 1: Logical Layout of VNS Retriever

Communication within the architecture necessitates the use of a standard format for queries. We use what we refer to as our canonical query form to define the logical structure of a query and its response. The purpose of the canonical form is to provide a common internal structure for use in mapping terms as displayed by a user interface into terms appropriate to diverse databases. For example, where MEDLINE's Elhill query language uses the indication "(AU)" to identify queries on author names, another local database in common use uses ".au" and the ANSI standard Z39.50 specification uses "ua". On the other hand, since there is no reason to be cryptic in displaying this information to the user, we display the term as "Author" (the substitution of some language other than English for these terms would be straightforward).

The canonical form also considers the type of data to be retrieved. Among the types that we intend to provide in addition to ASCII text are images (of numerous types) and possibly structured data such as those from gene sequence databases. The data recoverable by the MEDLINE Retriever are all textual data, so the only type currently supported is ASCII strings.

The query and presentation terms we use are based on the Z39.50 standard.

A physical view of the VNS Retriever architecture is found in Figure 2. Numerous invocations of the various user interfaces communicate one-to-one with one of several query managers running at the institution. Each query manager serves a specific work group of users related to some administrative entity within the institution; such an entity would typically be a grant-funded research group.

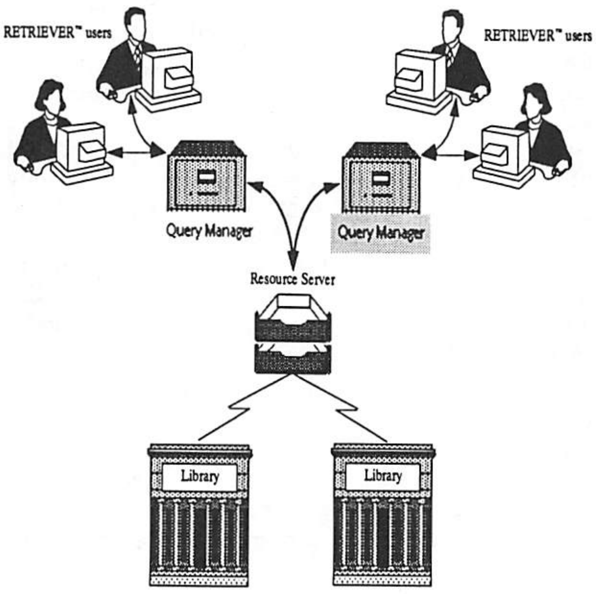


Figure 2: Physical View of the VNS Retriever

Each query manager in turn communicates with the single institution-wide resource server; the resource server is then able to invoke one or more of several resource query engines, each of which may serve more than one database.

The resource server and query manager are the two central pieces of the architecture. We shall describe them first, and then discuss interprocess communication within the system. Since the user interface and resource query engines are implementation dependent, we defer discussion of them to our detailed description of the implementation of the MEDLINE Retriever.

The Resource Server

The server agent is called the resource server. Its administrative role is to centralize institution-wide access control and accounting. This permits us to address a pet peeve in our user community; that is, the need to enter a user id and password for every invocation of a controlled-access program. Access to the resource server is restricted to a set of authorized machines. Access to individual machines is controlled by the machine's protection mechanisms. Accesses from specified machines can be further restricted to a subset of the available resources by means of a resource management table. Our design anticipates that installation on our campus would have a single resource server providing service to the entire organization.

The resource server's role in database retrieval is to provide a common interface to the diverse set of resources that it can support, allowing a single user interface to access all the supported resources. To this end, it supports requests to enumerate the databases it serves and to list the query and presentation terms valid for each. When the resource server receives a search request, it stores the request in its internal database, which is a simple tree with branches for each work group. The server then analyzes the request to determine which database to query. At this point it performs whatever accounting and access control are appropriate to the database, and then translates authorized requests into the appropriate terms for that database. Finally, the query engine for that database is invoked through a set of library routines that launch the query engine in a synchronous exchange, and then retrieve its results when they return asynchronously some time later.

After a query engine returns a response, the resource server notifies the appropriate client query manager. When the client has acquired the result data and acknowledged its receipt, the query and the response are deleted from the local database. Any cost accounting to be done for a given database is performed at this point.

The Query Manager

The user agent is called the query manager. Its administrative role is to provide middle- to long-term storage of queries and results for its work group, thus placing the responsibility for disk utilization on each work group, which runs its own query manager.

The query manager provides the user interface with a set of mappings to translate queries from user display form into canonical form, and to translate results from canonical form to the user-preferred display. It also provides the user interface with the list of available database resources and their query terms. The query manager connects to the resource server to submit queries. The query manager holds responses it receives from the resource server until they are deleted by request of the end user via whatever front end the user may use.

In addition to its intermediary role in database retrieval, the query manager maintains the user profiles for its clientele. These profiles describe what information the user wishes to retrieve as a default, and how it should appear. They also describe translations, as well as the filters and destinations to be used for exporting results to other places.

The query manager maintains little state concerning on-demand search requests. The query manager knows when it has issued a search request to the resource server, and provides a means to delete outstanding requests It depends on the resource server to maintain other information about the state of an on-demand search. The query manager does, however, keep track of scheduled queries in more detail. Selective Dissemination of Information (SDI) is the name that the NLM gives to searches that are stored centrally at the NLM database for incremental execution on a regular basis, for example, monthly when the database is updated. The fact that the query manager permits query scheduling makes the storage of SDI queries at the query manager possible, reducing the burden on the central database and opening the door for regular scheduling of intelligent wide-area database searches when such searches become viable. It is the query manager's job to queue scheduled queries and submit them to the resource server at the appropriate time.

Interprocess Communication

There are several levels of interprocess communication within VNS Retriever. Communication between the upper levels of the system relies on the use of a canonical form for the description of query terms and results. We shall discuss in turn the communications between the query manager and its user interfaces, between the query manager and the resource server, and between the resource server and its query engines.

The Query Manager Interface

The query manager communicates with its user interfaces through an application program interface whose standard queries include user profile handlers, search manipulation, and administration, as well as a search-result callback.

The functions Query-Profile and Modify-Profile allow the user to list and alter default settings for the Z39.50 terms Large-Set-Lower-Bound and Small-Set-Upper-Bound (the maximum number of matching records for which the user deems it is worth returning any data, and the largest number of records to be returned from a successful search, respectively) as well as desired presentation terms to be returned by a search, such as author, title, journal name, etc. This is also the place in which the cost center for charge accounting is specified. To facilitate exporting result data from VNS Retriever to other programs, in particular the VNS, the query manager supports the functions Query-Export and Modify-Export. These functions allow the user to manipulate defaults for data export from VNS Retriever. Groups also carry default settings for profile and export defaults, so that a new user automatically adopts the standards of the group to which it is added.

The function List-Searches returns a list of searches currently maintained for the user by the query manager, and List-Results returns the results for a given search, if they are available. Searches can also be renamed or deleted. A "force" flag in the delete function allows a search to be deleted irrespective of whether it is currently outstanding at the resource server.

The function Submit-Search has three options, submit on demand, submit scheduled, and submit to save. Saving a search simply records the contents of the query at the query manager so that it may be retrieved for later modification and submittal. A search may be scheduled for a one-time search at a later time, or for regular submittal at a restricted set of intervals such as hourly, daily or monthly (although there is no point at the current time in searching daily or more frequently, because updates to MEDLINE are not that frequent, it is our intention to incorporate local databases such as colloquium notices, for which more frequent regular search *would* be appropriate).

Status queries provide information on the current state and availability of results for the named search. Administration requests include the ability to add or delete users or groups, to request that the query manager quit gracefully, and to request changes in the type of information recorded in the log file.

Since the return of results from the resource server to the query manager is an asynchronous event, a user interface can register a result callback so that it can be informed in a timely fashion of the return of a search, without the need to poll the query manager.

The Resource Server Interface

The query manager behaves as a client of the resource server, sending synchronous requests. The resource server recognizes three basic types of request: search, status, and administration.

The resource server's search protocol is based loosely on the Z39.50 standard. The Init and Init-Response Application Protocol Data Units (APDU's) are used to establish a connection between a query manager and the resource server. The Search APDU is used to initiate a search, and a Search-Response APDU is used to acknowledge the successful receipt and launch of the search, using a return code not specified by the Z39.50 standard to indicate this initial success. Because the initial response to the search APDU is effectively an acknowledgment of the request, the query manager is free to carry on with other tasks instead of blocking synchronously on the search. This deviation from Z39.50 protocol enables the query manager to multiplex on several clients potentially issuing multiple requests per client.

Upon receipt of search results from the query engine, the resource server issues an announcement to the client query manager by means of an asynchronous Search-Response APDU. A fixed-size portion of the result data is piggy-backed with this announcement. This piggy-back packet is sufficient to return the entire response in many cases. Should the response contain more data than will fit in this announcement, an indicator is set to show that more data remain to be retrieved. It is then up to the query manager to request and retrieve the remainder of the data by means of Present and Present-Response APDU's.

There are certain resource server functions that lie outside the Z39.50 protocol. These include administrative functions, including addition and deletion of users and groups (although the identification of a user is not strictly necessary to the resource server's function, it provides a parallel structure to that found in the query manager's internals).

Status queries return information on the state of the server, as well as specific information about groups users, or queries. An additional status query returns the list of resources supported by the resource server. In our initial implementation, this is a trivial response, since the MEDLINE Retriever supports only one database.

The Query Engine Interface

Communication between resource server and query engine is by means of library calls. It is the query engine's responsibility to provide all communication with the database(s) it serves. Each query engine interface requires a set of translations from the canonical form to the query engine's native query language (although the query engine's language may simply be the query language of the database it supports, this need not be the case for a more general query engine, such as the Knowbot query engine), as well as the routines to initialize and launch a query engine instance; to initialize a response retrieval and to drive data retrieval; and to destroy the query engine instance.

The MEDLINE Retriever

The first instantiation of VNS Retriever has been the MEDLINE Retriever. This system has been tested and used by members of the Baylor research community since February, 1992. The MEDLINE Retriever takes advantage of the ABIDE gateway and Knowbot Operating Environment developed by David Ely of CNRI [8]. The Knowbot query engine is used to provide access to MEDLARS, the host for the National Library of Medicine's on-line bibliographic database. We shall first briefly describe the user interfaces for the system. Then we shall discuss the internals of the resource server and query manager and briefly describe the behavior of the Knowbot query engine.

User Interface

The user interface for the MEDLINE retriever does not reflect all of the generality described for the VNS Retriever architecture, but it provides a basis from which to understand the system, as well as a fully functional MEDLINE query tool that is easily integrable into other environments such as the VNS.

At the top layer we have demonstrated several user interfaces. There is a command-line interface that reads queries from files created by the user, and then formats the results to standard output. Another interface uses a simple form-driven X-Windows front end to the command-line interface, and formats the results into another window that permits simple user browsing through the returned citations.

The interface to which we have paid the most attention is a menu-driven X-Windows interface written using the Motif toolkit. This front end facilitates more sophisticated handling of queries and results, including creating queries using MeSH, querying for the status of outstanding searches and exporting selected citations to the Virtual Notebook System, as well as to order photocopies of desired citations direct from the Houston Academy of Medicine-Texas Medical Center Library. This interface is described in more detail elsewhere [9].

Server Structure

Both the resource server and the query manager maintain a mirror of their internal trees on stable storage. This allows both servers to maintain their state across invocations by reading their respective storage trees found on disk to build their internal trees at start-up. When the resource server initializes, it examines this tree for searches that have not yet received a reply from their query engines; it resubmits such searches, since the return address for the query engine instance associated with the old request is now invalid. When the query manager initializes, it builds a queue of scheduled and regular searches that it finds on disk, and sets a time-out for the shortest interval found among those queued.

We have chosen to store the data on disk in ASCII files, although we could substantially reduce storage by use of binary data. This permits visual inspection of the storage tree to help detect system problems, and simplifies debugging. Duplicating the information on disk requires frequent file writes that certainly have some impact on performance. We could improve this performance in at least one respect: Our current implementation employs a

UNIX file system, creating a directory for each group, user, and query. Replacing this mechanism with a common database manager such as *ndbm* would probably significantly improve our performance, at the expense of losing the ability to inspect the storage tree visually. This is a problem that we will address again when scaling becomes a greater issue than it is now.

Communication Protocols

Figure 3 shows the interprocess communication paths within the MEDLINE Retriever. Each level of the system employs a conventional client-server model for most transactions. At each level, an upcall or callback is also necessary to provide asynchronous notification. The query manager uses SunRPC [10]. Below that, TCP/IP network sockets are used.

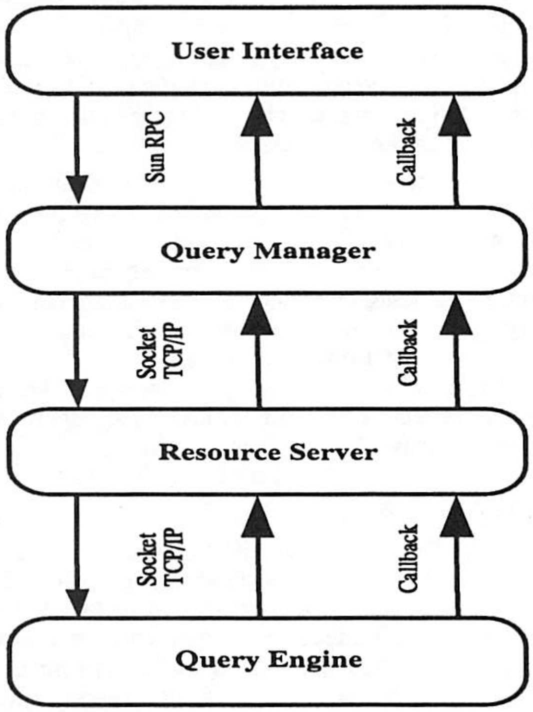


Figure 3: MEDLINE Retriever Interprocess Communication

The Resource Server

The connection between query manager and resource server employs BSD network sockets, using TCP/IP. The resource server opens listening sockets on one front end port and one back end port. The driving loop maintains one global socket, as well as one socket per active client, one initiator socket per query engine, plus one socket per active query.

When a query manager initializes a connection to the resource server, the resource server sets a short time-out after replying. When the time-out expires, the resource server examines its internal tree for outstanding responses belonging to the newly connected query manager. Any such responses have never before been successfully reported to the query manager, so at this point they are dispatched to the query manager with the asynchronous Send-ResponseAPDU. This greatly simplifies connection initialization.

The Query Manager

The query manager's internal structure has much in common with the resource server. In fact, their executable code links a common library for manipulating the internal storage tree.

Connections between the query manager and the user interfaces utilize SunRPC for interprocess communication. To some extent the use of two different implementations for our communications protocols represents an experiment. We chose to use SunRPC in order to reduce the overhead caused by the use of TCP/IP, but let SunRPC simplify the handling of UDP connections. As a serendipitous effect, this permits us transport independent communication.

Although other RPC mechanisms are available, we chose SunRPC without examining other alternatives simply because it was freely available on our initial target platforms. We have found SunRPC to be congenial, although development was more difficult than it might have been because there is no checking for null string pointers in the external data representation library.

Because the query manager is effectively in the middle of a pipe, acting as a server on one side and as a client on the other, it cannot use the SunRPC svc_run() call to drive its loop. The query manager opens its front end by means of SunRPC initialization, and also opens a back end socket with which to connect to the resource server. The driving loop simply deals with the SunRPC communications first, thereafter responding to any asynchronous responses from the resource server.

Because of the differences in treatment of SunRPC between the procedural model of a "normal" UNIX program and the event-driven model of an X Windows application, there are two slightly different application program interfaces available that handle the two cases.

To handle regularly scheduled SDI searches, the query manager builds a queue, and sets an alarm for the shortest interval found among the scheduled searches. The action taken at alarm time is simply to set a flag to indicate the elapse of the interval. When the query manager falls to the bottom of its driving loop, it tests this flag and invokes the queue handler if set. This avoids the possibility of inconsistency in the internal tree, because there is no access to internal data structures during the execution of the asynchronous alarm handler.

Help Server

Additionally, the system provides a help service. On-line help is available for most objects in the graphic user interface. In order to provide maximum flexibility and host independence, this help is provided by remote procedure calls to a help service. Help requests are keyed by program and function name. The contents of the file corresponding to the key are returned to the requester as an ASCII text object. The requesting interface can display the information as it sees fit (button-specific helpfiles probably make sense for only one interface, of course). This service is isolated from the query manager in order to permit the support of multiple related programs out of the same help service.

Query Engines

Although we anticipate several distinct query engines, we currently support only the Knowbot Operating Environment.

The Knowbot Operating Environment

In the specific case of the MEDLINE Retriever, the query engine is provided by the pioneering work done by David Ely at CNRI in creating the ABIDE gateway to MEDLARS and the Knowbot Operating Environment. This gateway receives appropriately constructed Knowbot queries, validates and executes them and return the results to the Knowbot constructor that sent them.

Our collaboration with CNRI on the VNS Retriever project has provided CNRI with an opportunity to shake down the concepts involved in the Knowbot Operating Environment. At the same time, it has allowed us to work directly on creating an environment for campus use that could be replicated in other campus environments where access to centralized databases is important.

The ABIDE gateway was originally created by CNRI as a demonstration project for the NLM. Its user interface closely duplicated the functionality of the MS/DOS version of Grateful Med, but provided no query management features or asynchronous access. Through this cooperative effort, we were able to make use of the back end functionality while enhancing the power of the user interface.

The ABIDE gateway runs on a Sun-4 machine located at the NLM. It serves to connect the IBM environment on which the MEDLINE database is operated to the Internet.

The transactions between the resource server (via the Knowbot library) and the ABIDE Gateway use TCP/IP. As described above, the Knowbot query engine is launched in one synchronous action. The launch of the Knowbot query includes a return address to which the ABIDE gateway will subsequently respond asynchronously. Each query from a given MEDLINE account is queued at the ABIDE gateway and performed sequentially.

Our use of the Knowbot Operating Environment does not exploit its ability to return data to a receiver on a different host from the launcher. This design decision is based on our desire to control access and perform accounting centrally.

Other Query Engines

In the case of our first implementation, which involves internet communication, initialization and launch comprise the first step of the search process, and at this point the resource server reports a successful launch to its client. Thereafter, the server responds to activity on the retrieval socket by invoking the query engine's data-retrieval function. Anticipated future query engines that involve local area access may have essentially null query retrieval functions, since it is believed that data response will be quick enough to incorporate in the launch phase. Such queries will also bypass the stage involving recording on stable storage for the sake of speed.

Among the query engines that we are considering for addition are a Wide Area Information Service (WAIS) engine, and an RPC-based Unified Medical Language System (UMLS) engine [11].

Administration

Administration of this system occurs at two levels. The Query Manager and the Resource Server each have separate administrative requirements and can be administrated separately as needed. The architecture provides for one Resource Server per site and one or more Query Managers.

Resource Server Administration

Administration of the Resource Server involves checking accounting information for each request submitted. In order to give system managers at different sites local control over how the resource server does accounting, we provide a hook for a remote procedure call to an external server. This server simply returns zero for access accepted and non-zero for access rejected. The server does its permission checking by reading a script written by the system manager to suit the needs of the individual institution.

Additionally, the Resource Server verifies information supplied by the Query Manager about the group or the user submitting the query to insure that the group or user is authorized to make use of the resource to which the query is directed. If the user or group is authorized, the query is processed. If not, the Resource Server returns an **ACCESS DENIED** reply in response to the query. Password management and all other resource access information is stored in the Resource Server access database.

Query Manager Administration

Management of the Query Manager centers on the disposition and storage of queries and query responses. Most of the management of the Query

Manager occurs with each user authorized to make use of it. Individual users enter queries and determine how to store responses.

However, there are certain matters that require an administrator. Authorizing users to make use of a particular query manager, removing old users (including purging any queries or query responses created by those users) and providing a default user profile must be done by the Query Manager Administrator.

Performance and Evaluation

The performance of the MEDLINE Retriever has impressed our user group. Response times are good from half a continent away. Although the system has many good qualities, there is ample room for improvement.

System Performance

In a battery of timing tests we ran using the command-line user interface, we measured both the time required by the resource server, and the round trip time from command-line user interface to MEDLARS and back. At the resource server, the average response time (from initialization of query engine instance to completion of data retrieval) was 11.7 seconds for a search that returned 10 citations (author, title, and source only) from a query involving text search of two words. The average response time was 18.5 seconds for the same search to return 200 citations, which required transmission of about 70 Kilobytes of data (we impose a limit of 200 on Small-Set-Upper-Bound, the maximum number of records to return, which is a compromise between the belief that we share with some librarians that one generally does not wish to examine any results at all until one has pared the result set down to a large handful, and the use-patterns and desires of many of our scientific users, who do not mind browsing even hundreds of titles in search of relevant citations).

The round-trip times at the command line interface were less satisfying. The average response time in the case of returning 10 citations was 15.9 seconds, an average four second overhead for the system architecture. Returning 200 citations took an average of 27.4 seconds, an average 9 second overhead. Naturally, virtually all of this overhead is clock time, not CPU; unfortunately that is what the user perceives. Command line response time is affected by the times incurred by both query manager and resource server in polling their sockets during select() calls, as well as time required to update status files. We intend to eliminate the causes of this excessive overhead in the near future.

During normal use, the average response time for a successful search was 16 seconds. Most searches (292 of 324) required less than 30 seconds to return. Of those, the average was 9.6 seconds. The longest successful search required 5 minutes 23 seconds.

The average time required to initialize a query engine instance was 1.2 seconds, which involves opening a socket on the local host and establishing a connection to the ABIDE gateway. The maximum initialization time was 9.9 seconds.

The average cost of these searches as reported by the ABIDE gateway was $2.30. A simple search that returns little or no data costs around a dime, but any attempt to retrieve larger amounts of data, either to retrieve abstracts, at around 3 Kilobytes apiece, or to retrieve numerous citations, causes the charge to increase rapidly. This has raised the issue of how to account for MEDLINE searches. The cost schedule that is currently used by the NLM is perceived as prohibitive by our users.

Evaluation of the System

Overall, we are pleased with the results of our design. One question that remains to be resolved is whether a single resource server will scale well for full-scale use at a large institution. Baylor has about forty specialized research centers comprising around eight hundred laboratories with several thousand scientists participating in research, and another several hundred involved primarily in education. A more detailed analysis of response time under a heavy load is needed to determine whether a single resource server can in fact support the whole community at Baylor.

Another potential problem with respect to MEDLINE specifically is that queries issued on a single account are queued. Since our current practice is to issue all queries on one account, this can adversely affect users' response times. Although our actual deployment policy is as yet undetermined, we may choose to employ multiple MEDLINE accounts to circumvent this situation if it actually causes difficulty.

With regard to our decision to logically isolate our resource server from the query engines, there are advantages and disadvantages. On the one hand, much of the knowledge necessary to access diverse databases exists already in the Knowbot Operating Environment, and our logical separation from that environment represents some duplication of effort. On the other hand, use of our own canonical query form permits us to translate to and employ other query engines with minimal effort; it also permits us to consider the use of lighter weight mechanisms for inherently small local databases, such as "bcm.seminars", the college's electronic colloquium bulletin board. Overall, this logical isolation provides a flexibility that is a greater boon than a hindrance.

We chose to use our own canonical form in preference to SQL partly from convenience, and because many resources which we wish to access are not in a relational form for which SQL is well-suited. Similarly, we do not simply employ a free-text-searching algorithm in order to take advantage of the structured nature of many common resources.

As for our implementation, there are several things we will likely change in a future release of the system. First, we would like to replace the existing socket-based transport used in the resource server with an RPC model. This would take advantage of the RPC-based communication protocol that already exists in the query manager. Most of the fundamental requests in both servers are identical in structure, if not in effect, so the parallelism of design would be reflected in reuse of code (the original specification of the query manager used socket-based communications for that reason).

We had chosen the socket model for the resource server's communications in part to take advantage of the Z39.50 library available from Thinking Machines Corporation's WAIS project [12]. Since the Z39.50 protocol does not handle the problem of asynchrony we have mentioned earlier, there is some disadvantage to adhering to that model. There is, however, substantial benefit to the natural way in which the SunRPC package wraps around our canonical forms.

Finally, it might behoove us to take responsibilities away from the user interface. Many of the functions that it performs would be better performed by small utility routines that it might invoke in the course of its execution.

We are pleased with the use of SunRPC, although more gracious handling of null string pointers in the external data representation code would be welcome.

Although we considered the use of the Sun lightweight process library, the benefit that it would have provided us in terms of imposing a logically multi-threaded design was rendered null by the problems to be overcome in compensating for the fact that a single lightweight thread blocking on I/O blocks the entire process. We also had difficulties with incompatibility in the memory allocation.

Related Work

The NLM has recently promulgated an Internet-capable version of Grateful Med. This is an excellent system for its purpose, which is to provide user-friendly support for MEDLINE access; however, it exists of itself, and does not integrate well with the goals of IAIMS at Baylor.

We also wish to reiterate our respect for the work of CNRI, whose Knowbot Operating Environment is the foundation of our link to MEDLARS at NLM. Their coop- eration has contributed markedly to our success. The Knowbot query language provides a single Z39.50-like query interface to each of the databases supported by ABIDE. It is an ideal back end for our work, because there is great potential for the expansion of its powers in order to expand ours.

MCC's Carnot project is a vastly more ambitious system that incorporates both database retrieval and update, employing a transaction shell called Rosette that uses the Actor model for organizing asynchronous distributed database accesses [13]. The development of a Carnot query engine could significantly extend the power of our system.

The WAIS project takes a different view of wide area data retrieval, employing full-text search instead of keyword indexing. This is a technology ideally suited to the parallelism of the Connection Machine for which it was originally designed. The user interface to a WAIS service could be subsumed by our existing user interface.

The University of Minnesota's **internet gopher** project implements a document delivery service. It allows users access to various types of information residing on multiple computers in a seamless fashion. The interfaces for gopher use a hierarchy of menus. Each menu item may provide access to other menus, files, or gateways to other types of information servers includes the WAIS environment described above.

Future Work

The design of MEDLINE Retriever has laid the foundation for several further development efforts. These include the augmentation of the powers of the back end, closer integration with the VNS, and expanding support for alternative data types.

Back End Support

We intend to expand both the number of query engines available to the resource server, as well as to include support for several other resources. Among the resources we hope to add are Current Contents, as well as a service to index colloquium announcements and other institutional events. Standing queries for these latter resources can be established to report upcoming events of interest to each user. We hope to deploy a local ABIDE gateway in order to experiment with the use of Knowbot scripts for access to these resources.

Other resources that we wish to add suggest the construction of new query engines. One of these is a UMLS server to enhance the power of the user interface to the MEDLINE Retriever. Another would be a WAIS query engine.

Closer Integration with VNS

With the ongoing development of version 3.0 of the VNS, we have the opportunity to incorporate wide area hypermedia into the VNS. We have dubbed this new model VNS-STAR. The VNS-

STAR model distinguishes between the page, VNS's traditional vehicle for data organization, and the STAR-PAGE. Objects on a STAR-PAGE contain not data, but the reference information necessary to recover a specific dataset from anywhere on the internet, thus trading bandwidth for storage, and increasing the timeliness of data displayed through the VNS.

Another area of interest is adapting VNS Retriever for the retrieval and display of alternative data types. This is closely tied to the object orientation in VNS 3.0. VNS Retriever will initially recognize the types defined by the VNS, particularly images. Any further expansion of VNS Retriever's type set can be done through the user-definable objects of the VNS. Because of VNS Retriever's use of high-speed networking, it is well suited to take advantage of imagery and other higher-volume data that will inevitably become part of the typical bibliographic resource.

Conclusion

The VNS Retriever has served a useful role in helping us develop an architecture to systemize access to bibliographic and other resources. It is a necessary step towards developing more intelligent networked data retrieval tools. As a tool in its own right, it is performing satisfactorily for the BCM user community, while at the same time, it is serving to illuminate the challenges that lie ahead in our development of more useful tools for the typical researcher.

VNS Retriever is representative of many of our efforts. Even as we attempt to create an overall approach to addressing classes of problems, our user community benefits from the application of a useful tool which addresses a specific need. The VNS Retriever is paving the way for more facile access to the ever-expanding collection of internetworked resources.

Acknowledgments

Support for this work was provided by The National Library of Medicine (N01-LM-1-3516 and USPH-02-G08-LM04905).

Thanks are due to Ed Sequeira of the NLM, and to David Ely and Vint Cerf of CNRI for making our use of ABIDE possible. In addition, we wish to thank Tony Gorry and the rest of the Baylor IAIMS group for their ideas and encouragement. Thanks to Cindy Petermann and Barry Meyer of the group for their helpful comments on the manuscript.

For more information on the Knowbot Operating Environment, contact CNRI electronically at: knowbot_info@nri.reston.va.us

References

[1] A. M. Burger, et. al. "The Virtual Notebook System," *Proceedings of the Hypertext '91 Conference,* ACM, 1991.

[2] G. A. Gorry, et al., "Computer Support for Biomedical Work Groups," *Proceedings of the Conference on Computer Supported Cooperative Work,* September 1988.

[3] G. A. Gorry, et al., "The Virtual Notebook System: An Architecture for Collaboration," *Journal of Organizational Computing,* Volume 1 Number 3, 1992.

[4] R. Scheifler and J. Gettys, *The X Window System,* Digital Press, 1988.

[5] Open Software Foundation, *OSF/Motif Style Guide,* Prentice-Hall, 1990.

[6] R. E. Kahn and V. Cerf, *An Open Architecture for a Digital Library System and a Plan for its Development, The Digital Library Project, Volume 1: The World of Knowbots,* Corporation for National Reserach Initiatives, Reston, Virgina, 1988.

[7] B. Shearer, L. McCann, L. and W. J. Crump, "Grateful Med: Getting Started," *Journal of the American Board of Family Practice,* Volume 3, Number 1, pp. 35-38, January-March, 1990.

[8] David Ely, *An Overview of the ABIDE Gateway System,* Technical Report TR-91-1, Corporation for National Research Initiatives, Reston, Virginia, 1991.

[9] J. Fowler, K. B. Long, and S. Barber "The MEDLINE Retriever," Submitted, *16th Annual Symposium on Computer Applications in Medical Care,* Baltimore, Maryland, November 1992.

[10] Sun MicroSystems, Inc., *RPC: Remote Procedure Call Protocol Specificaion Version 2; RFC 1057,*Network Center (NIC) at SRI International, Menlo Park, California, June 1988.

[11] S. Barber, J. Fowler, K. B. Long, R. Dargahi, B. Meyer, "Integrating UMLS into VNS Retriever," Submitted, *16th Annual Symposium on Computer Applications in Medical Care,* Baltimore, Maryland, November 1992.

[12] Brewster Kahle, *Wide Area Information Concepts,* Thinking Machines Corporation, Technical Memo DR-89-1.

[13] Microelectronics and Computer Technology Corporation, *MCC Carnot Project Description,* Austin, Texas, 1991.

Author Information

Kevin Long is Director of IAIMS Development at Baylor College of Medicine. A graduate of Rice University, Kevin has been involved in the Integrated Academic Information Management System initiative at Baylor since 1985, and is Vice President of The ForeFront Group, Inc., a

technology-transfer company which distributes the VNS outside of BCM. Reach him via U. S. Mail at Baylor College of Medicine; One Baylor Plaza; VPIT; Houston, TX 77030-3498. Reach him electronically at klong@bcm.tmc.edu.

Jerry Fowler has been a member of the technical staff of Integrated Academic Information Management System Development Group at Baylor since 1991. He holds bachelor's degrees in math and music from the University of Oklahoma and a master of science in computer science from Rice University. His U. S. Mail address is IAIMS Development, Baylor College of Medicine; One Baylor Plaza; Houston, TX 77030-3498. His electronic home is gfowler@bcm.tmc.edu.

Stan Barber is Director of Networking and Systems Support at Baylor College of Medicine. A graduate of Rice University, Stan has been involved in the Integrated Academic Information Management System initiative at Baylor since 1986. Stan also supports ongoing development of **rn,** the popular news reader program originally developed by Larry Wall. Reach him via U. S. Mail at Baylor College of Medicine; One Baylor Plaza; Houston, TX 77030-3498. Reach him electronically at sob@bcm.tmc.edu.

InterNetNews: Usenet transport for Internet sites

Rich Salz – Open Software Foundation

ABSTRACT

NNTP, the Network News Transfer Protocol, has been labelled the most widely implemented elective protocol in the Internet. The growth of the Internet has meant more sites exchanging NNTP data. While the explosive growth in Usenet traffic places demands on all sites, the goal of fast network access puts particular demands on NNTP hosts.

InterNetNews is an implementation of the Usenet transport layer designed to address this situation. It replaces the standard UNIX server architecture with a single long-running server that handles all incoming connections. It has proven to be quite successful, providing quick and efficient news transfer.

Introduction

Usenet is a distributed bulletin board system, built as a logical network on top of other networks and connections. By design, messages resemble standard Internet electronic mail messages as defined in RFC822 [Crocker82]. The Usenet message format is described in RFC1036 [Adams87]. This defines some additional headers. It also limits the values of some of the standard headers as well as giving some of them special semantics.

Newsgroups are the classification system of Usenet. The required Newsgroups header specifies where a message, or article, should be filed upon reception. Sites are free to carry whatever groups they want. Most sites carry the core set of so-called "mainstream" groups. There are currently about 730 of these groups, and one or two new ones is created every week.

Messages generated at a site are sent to the site's "neighbors" who process them and relay them to their neighbors, and so on. Sites can be interconnected – indeed, on the Internet, this is quite common. See Figure 1.

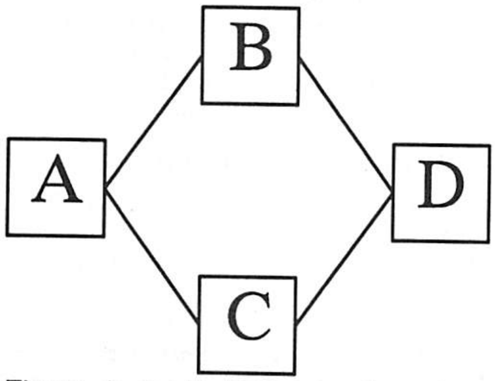


Figure 1: Small Usenet topology (all links are two-way).

The Path header is used to prevent message loops. For example, an article written at *A* could get sent to *B*, *D*, *C*, and then back to *A*. Before propagating an article, a site prepends its own name to the Path header. Before propagating an article to a site, the receiving host checks to make sure that the site that would receive the article does not appear in the Path line. For example, when the article arrived at site *C*, the Path would contain *A!B!D*, so site *C* would know not to send the article to *A*.

Sites also keep a record of the Message-ID's of all articles they currently have. If *D* receives an article from *B*, it will reject the article if *C* offers it later. For self-protection, most sites keep a record of recent articles that they no longer have. This is very useful when another site dumps a (usually quite large) batch of old news back out to Usenet.

For the past few years, the amount of data generated by Usenet sites has been doubling every year. A site that receives all the mainstream groups is receiving over 17 megabytes a day spread out over 11,000 articles [Adams92]. About 20% of the data is article headers, and while all of them must be scanned only half of it is must be processed by the Usenet software.[1]

The number of sites participating in Usenet has been growing almost as quickly. Based on articles his site receives and survey data sent in by participating sites, Brian Reid estimates that there are 36,000 sites with 1.4 million participants [Reid91]. A "sendsys" message to the "inet" distribution in June of 1989 received about 200 replies in the first twenty-four hours. A year later, nearly 700 replies were received. (Sendsys is a special article that asks all receiving sites to send back an email message, usually without human intervention; by convention, inet is primarily the set of sites on the Internet.)

[1]Yes, this means that, as far as the software is concerned, Usenet is over 90% noise.

The NNTP protocol is defined in Internet RFC 977 [Kantor86] published in February, 1986. This was accompanied by the general public release of a reference implementation, also called "nntp." This has been the only NNTP implementation that is generally available to UNIX sites.

Usenet Software

In addition to InterNetNews, there are two major Usenet packages available for UNIX sites. All three share several common implementation details. A newsgroup name such as comp.foo is mapped to a directory comp/foo within a global spool directory. An article posted to a group is assigned a unique increasing number based on a file called the *active* file. If an article is posted to multiple groups, links are used so that only one copy of the data is kept. A *sys* file contains patterns describing what newsgroups the site wishes to receive, and how articles should be propagated. In most cases, this means that a record of the article is written to a "batchfile" that is processed off-line to do the actual sending.

The first Usenet package is called B News, also known as B2.11. The B news model is very simple: the program *rnews* is run to process each incoming article. Locking is used to make sure that only one *rnews* process tries to update the active file and history database. At one site that received over 15,000 articles per day, the locking would often fail so that 10 to 100 duplicates were not uncommon. Because each article is handled by a separate process, it is impossible to pre-calcuate or cache any useful data.

More importantly, file I/O had become a major bottleneck. A site that feeds 10 other sites does over 150,000 open/append/close operations on its batchfiles. It is generally agreed that B news cannot keep up with current Usenet volume; it is no longer being maintained, and its author has said more then once that the software should be considered "dead."

C News gets much better performance then B news by processing articles in batches [Collyer87]. The *relaynews* program is run several times a day to process all the articles that have been received since the last run. Since only one *relaynews* program is running, it is not necessary to do fine-grain locking of any of the supporting data files. More importantly, it can keep the entire active and sys file in memory. It can also use buffered I/O on its batchfiles, reducing the amount of system calls by one or two orders of magnitude.

An alpha version of C News was released in October, 1987. Within four years it surpassed B news in popularity, and there are now more sites running C News then ever ran B news.

From the beginning, the NNTP reference implementation was layered on top of the existing Usenet software: an article received from a remote NNTP peer was written to a temporary file and the appropriate *rnews* or *relaynews* program processed it. In order to avoid processing an article the system already has, it first does a lookup on the history database to see if the article exists. It soon became apparent that invoking *relaynews* for every article lost all of C News's speed gain, so the NNTP package was changed to write a set of articles into a batch, and offer the batch to *relaynews*.

When articles arrive faster then *relaynews* can process them, they must be spooled. If two sites (*B* and *C* in the previous examples) both offer a third site (*D*) the same article at the "same time" then an extra copy will be spooled, only to be rejected when it is processed, wasting disk space; this problem multiplies as the number of incoming sites increases.[2] To alleviate this problem, most sites run Paul Vixie's *msgidd*, a daemon that keeps a memory-resident list of article Message-ID's offered within the last 24 hours. The NNTP server is modified so that it tells this daemon all of the articles that it is handing to Usenet and queries the daemon before telling the remote site that it needs the article. This is not a perfect solution – if the first, spooled, copy of the article is lost or corrupted, the site will likely never be offered the article after the *msgidd* cache entry has expired. Going further, *msgidd* is work-around for a problem inherent in the current software architecture.

Other problems, while not as severe, lead to the conclusion that a new implementation of Usenet is needed for Internet sites. For example:

- Since all articles are spooled, *relaynews* cannot tell the NNTP server the ultimate disposition of the article, and the server cannot tell its peer at the other end of the wire. This hides transmission problems. For example, a site tracing the communication has no way of finding out an article was rejected because the remote site does not receive that particular set of newsgroups.
- The NNTP reference implementation is showing signs of age. Maintaining the server is becoming a maintenance nightmare; over one-tenth of its 6,800 lines are *#ifdef*-related.
- All articles are written to disk at extra time. Disks are getting bigger, but not faster, while CPU's, memory, and networks are.

InterNetNews architecture

There are four key programs in the InterNetNews package (see Figure 2):

Innd is the principal news server for incoming newsfeeds;

[2]This is quite common for Internet sites, where redundant fast newsfeeds are common and where many Usenet administrators seem to be avid players of the "exchange news with as many other people as possible" game.

innxmit reads a file identifying articles and offers them to another site;
ctlinnd sends control commands to *innd*;
nnrpd is an NNTP server oriented for newsreaders.

Of these programs, the most important is *innd*. We first mention enough of its architecture to give a context for the other programs, and then discuss its design and features in more detail at the end of this section.

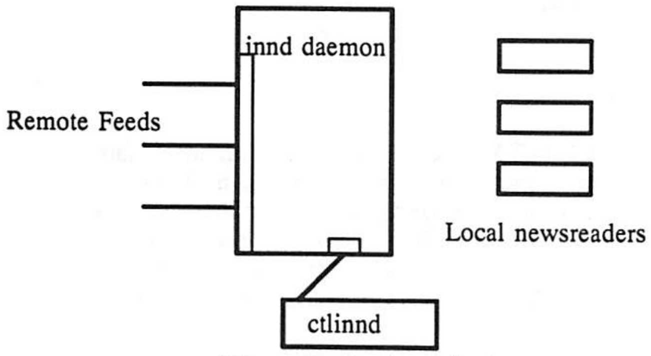


Figure 2: Innd architecture

Innd is a single daemon that receives all incoming articles, files them, and queues them up for propagation. It waits for connections on the NNTP port. When a connection is received, a *getpeername*(2) is done. If the host is not in an access file, then an *nnrpd* process is spawned with the connection tied to its standard input and output.[3] It is worth noting that *nnrpd* is only about 3,500 lines of code, and 20% of them are for the "POST" command, used to verify the headers in a user's article. *Nnrpd* provides all NNTP commands except for "IHAVE" and an incomplete version of "NEWNEWS". On the other hand, it does provide extensions for pattern-matching on an article header and listing exactly what articles are in a group. The NNTP protocol seems to be a good example of the UNIX philosophy: it is small, general, and powerful and can be implemented in a very small program.

Articles are usually forwarded by having *innd* record the article in a "batchfile" which is processed by another program. For Internet sites, the *innxmit* program is used to offer articles to the host specified on its command line.[4] The input to *innxmit* is a set of lines containing a pathname to the article and its Message-ID. Since the Message-ID is in the batchfile, *innxmit* does not have to open the article and scan it before offering the article to the remote site. This can give significant savings if the remote site already has a percentage of the articles.

[3]Unlike other implementations, no single INN program implements the entire NNTP protocol.

[4]Other programs, like *nntplink*, are supported but not part of the INN distribution.

Until recently, *innxmit* used *writev* to send its data to the remote host. At start-up it filled a three-element *struct iovec* array with the following elements:

```
[0]  { ".", 1 };
[1]  { placeholder };
[2]  { "\r\n" }
```

To write a line, the *placeholder* was filled in with a pointer to the buffer, and its length, and a single *writev* was done, starting from either element zero or one. While this implementation was clever, and simpler then what was done elsewhere, it was not very fast. *Innxmit* now uses a 16k buffer and only does a *write* when the next line would not fit. This is also consistent with ideas used throughout the rest of INN: use the *read* and *write* system calls, referencing the data out of large buffers while avoiding the copying commonly done by the standard I/O library.

The *ctlinnd* program is used to tell the *innd* server to perform special tasks. It does this by communicating over a UNIX-domain datagram socket. The socket is behind a firewall directory that is mode 770, so that only members of the news administration group can send messages to it. It is a very small program that parses the first parameter in its command line and converts it to an internal command identifier. This identifier and the remaining parameters are sent to *innd* which processes the command, and sends back a formatted reply. For example, the following commands stops the server from accepting any new connections, adds a newsgroup, and then tells it to recompute the list of hosts that are authorized newsfeeds:

```
ctlinnd pause "Clearing out log files"
ctlinnd newgroup comp.sources.unix m \
        vixie@pa.dec.com
ctlinnd reload newsfeeds "Added OSF feed"
ctlinnd go ""
```

The text arguments are sent to *syslog*(8) for audit purposes.

The most commonly-used *ctlinnd* command is "flush." This directs the server to close the batchfile that is open for a site, and is typically used as follows:

```
mv batchfile batchfile.work
ctlinnd flush sitename
innxmit sitename batchfile.work
```

The flush command points out another difference between INN and other Usenet software. The B News *inews* program needed no external locking – files were opened and closed for a very short window, the time needed to process one article. The C News *relaynews* could be running for a longer period of time. The only way to get access to a batchfile is to either lock the entire news system, which is overkill for the desired task, or to rename the file and wait until the original name shows up again. The INN approach is more efficient and conceptually cleaner.

Innd Structure

When *innd* starts up it reads the active file into memory. An array of NEWSGROUP structures is created, one for each newsgroup, that contains the following elements:

```
char  *Name;         /* "comp.sources.unix */
char  *Dir;          /* "comp/sources/unix/" */
long  Last;          /* 0211 */
int   LastWidth;     /* 5 */
char  *LastString;   /* "00211..." */
char  *Rest;         /* "m\n..." */
int   SiteCount;     /* 1 */
SITE  **Sites;       /* defined below */
```

The C comments above show the data that would generated for the following line in the active file:

```
comp.sources.unix 00211 00202 m
```

The *Last* field specifies the name to be given to the next article in the group. The *LastString* element points into the in-memory copy of the file. This number is carefully formatted so that the file can be memory-mapped, or updated with a single *write*.

A hash table into the structure array is built, using a function provided by Chris Torek [Torek91]. The hash calculation is very simple, yet empirically it gives near-uniform distribution. The secondary key is the highest article number, so groups with the most traffic tend to be at the top of the bucket.

The INN equivalent of the *sys* file read next. An array of SITE structures is created, one for each site, that contains the following elements:

```
BOOL  Sendit;
char  FileFlags[10];
```

The *FileFlags* array specifies what information should be written to the site's data stream when it receives an article. The subscription list for the site is then parsed, and for all the newsgroup that it recives, the matching NEWSGROUP structure will contain a pointer to the SITE structure.

Using these two structures it is easy to step through how an article is propagated:

```
extern ARTDATA *art;
extern SITE *Sites, *LastSite;
extern int nSites;
char **pp;
SITE *sp;
NEWSGROUP *ng;
int i;

while (*pp) {
  ng = HashNewsgroup(*pp++);
  if (ng == NULL)
    continue;
  AssignName(ng);
  for (i = 0; i < ng->nSites; i++) {
    if (MeetsSiteCritera(ng->Sites[i], art))
      ng->Sites[i]->Sendit = TRUE;
  }
}
for (sp = Sites; sp < LastSite; sp++) {
  if (!sp->Sendit)
    continue;
  for (p = sp->FileFlags; *p; p++)
    switch (*p) {
    case 'm':
      /* Write Message-ID */
    case 'n':
      /* Write filename */
    ...
    }
}
```

The ARTDATA structure contains information about the current article such as its size, the host that sent it, and so on. The *MeetsSiteCriteria* function is an abstraction for the in-line tests that are done to see if an article really should be propagated to a site (e.g., checking the Path header as described above). *AssignName* is described below.

At its core, *innd* is an I/O scheduler that makes callbacks when *select*(2) has determined that there is activity on a descriptor. This is encapsulated in the CHANNEL structure, which has the following elements:

```
enum TYPE    Type;
enum STATE   State;
int          fd;
FUNCPTR      Reader;
FUNCPTR      WriteDone;
BUFFER       In;
BUFFER       Out;
```

The *Type* field is used for internal consistency checks. There four different types of channels – local-accept, remote-accept, local-control (used by *ctlinnd*) and NNTP connection. Each type is implemented in anywhere from 100 to 1200 lines of code. The *Reader* and *WriteDone* function pointers, and the *State* enumeration are used for protocol-specific data. For example, *State* field is used by the NNTP channel code to determine whether the site is sending an NNTP command or an article. The *BUFFER* datatype contains sized reusable I/O buffers that grow as needed.

At start time *innd* calls *getdtablesize*(2) to create an array of channels that can be directly indexed by descriptor.

The code to listen on the NNTP port is show in Figure 3. When a host connects to the NNTP port, *select*(2) will report activity on the descriptor and call *RemoteReader* which will accept the connection and possibly create fill in a new CHANNEL out of the resultant descriptor.

It took a bit of effort to write the callback loop so that it was fair – i.e., so that the lowest descriptors did not get priority treatment. The problem was complicated because other parts of the server can add and remove themselves from the *select*(2) read and write mask, as needed.

Once the NNTP channel has been created for a site, the server is ready to accept articles from that site. The reader function for NNTP channels reads as much data as is available from the descriptor. If it is in "get a command" state, it looks for a simple \r\n terminator; if it is in "reading an article" state, it looks for a "\r\n.\r\n" terminator. If not found, it just returns; the data will become available at some point. If the terminator is found, it processes the data in the buffer. For filing an article, this means cleaning up the NNTP text escapes, and calling the article abstraction to process the article.

Processing the article is the largest single routine in the server. The *AssignName* shown above increments the high-water mark for the newsgroup. If the article has already been written to disk, a link is made under the new name. (Symbolic links, if available, can be used if the spool area spans partitions.) If the article has not been written, a *struct iovec* array is set up as shown below, where vertical bars separate each iovec element:

```
First headers...
Path: |LOCAL_PATH_PREFIX|rest of path...
Second headers...
|XREF_HEADER|
|Article body.
```

This is a very fast way of writing the article to disk; it avoids extra memory copies, and is only possible because the entire article is kept in memory until the last moment.

Future work

RFC 977 follows the SMTP protocol for sending text: line are terminated with \r\n, a period is placed before all lines starting with a period, and data is terminated with a line consisting of a single period [Postel82]. *Innd* must scan the text of all articles it receives and convert them to standard UNIX format. On the transmission side, *innxmit* must read the articles a line at a time in order to add the extra data. If all newsreading is done via NNTP, then articles could be stored directly in NNTP format, and *innxmit* could read and write the article in two system calls. The *innd* gains would not be as dramatic, but tests show it would still be somewhat measurable.

There is no NNTP "TURN" command, so that a single connection cannot be used for bidirectional article transfer. Turn-around is very successful on UUCP over conventional phone lines, but seems of limited use on higher-performance network links. The SMTP protocol has had a "TURN" command since its inception, but it has received no practical use. Several people find the idea of adding outgoing transfer to *innd* attractive, since it is already structured for multi-host I/O and the idea of caching

```
int
RemoteReader(cp)
    CHANNEL *cp;
{
    int newfd;
    struct sockadr_in remote;
    int remsize;

    newfd = accept(cp->fd, &remote, &remsize);
    if (InAccessFile(remsocket))
        CHANcreate(newfd, TYPEnntp, STATEgetcmd, NCreader, NCwritedone);
    else {
        ForkAndExec("nnrpd", newfd);
        close(newfd);
    }
}

int
RemoteSetup()
{
    int fd;

    fd = GetSocketBoundToNNTPPort();
    CHANcreate(fd, TYPEremote, STATElisten, RemoteReader, NULL);
}
```

Figure 3: Listening on an NNTP port

recent articles in memory has its appeal. Adding outgoing transfer to *innd* would take a moderate effort.

Conclusions and Comparisons

The InterNetNews architecture works. Profiling a production installation for 24 hours showed that *open*(2) accounted for 10% of the run time. Since the server only does one *open*(2) per article, it is not clear if any other performance tuning is needed. The profiling overhead accounted for 5% of the runtime.

Several optimizations are available because there is only one process, and because it is always running. For example, avoiding duplicates is an integral part of the server. If a second site offers an article while a first site is sending it, the NNTP code will put the channel to "sleep" for a short while before replying to the second offer. This is usually enough time to have the first site finish sending the article, reducing the number of duplicates from hundreds to nearly none, with no external programs.

Since the server is always running, the system has a much smoother performance curve. As a result, it "feels" much faster to users.

Another unexpected benefit is that articles are accepted or rejected synchronously. A user can post an article, and by the time their posting agent has returned, it has been written to the spool directory and queued for remote transfer. If there is a problem such as having an illegal newsgroup specified, the user founds out immediately.

The design of the server seems to be very good, split into abstractions that are very independant. For example, sites have no knowledge of incomming NNTP connections. Using callbacks lets each portion of the server safely do I/O without worrying that it might affect other parts. Much of the Usenet processing becomes trivial when serialized, such as access to the history file.

The design has also led to a fairly small program: it is under 13,000 lines, and about 20% of them are comments. This compares favorable to the 7,400 lines in the equivalent C News program and the 7,600 lines in the NNTP reference implementation.

Availability

The INN package is freely redistributable, and is available for anonymous FTP from ftp.uu.net as ~ftp/news/inn.tar.Z. It is discussed in the Usenet newsgroups news.software.nntp and news.software.b.

References

[Adams87] Rick Adams, Mark Horton, *Standard for Interchange of USENET Messages*, Request For Comments 1036, Marina del Rey, CA: Information Sciences Institute, 1987.

[Adams92] Rick Adams, *Total traffic through uunet for the last 2 weeks*, Usenet message <<1992Apr8.193050.8963@uunet.uu.net> in *news.lists*, April, 1992.

[Collyer87] Geoff Collyer and Henry Spencer, *News Need not be Slow*, Usenix Winter Conference, 1987.

[Crocker82] David H. Crocker, *Standard for the Format of ARPA Internet Text Messages*, Request For Comments 822, Marina del Rey, CA: Information Sciences Institute, 1982.

[Kantor86] Brian Kantor, Phil Lapsley, *Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News*, Request for Comments 977, Marina del Rey, CA: Information Sciences Institute, 1986.

[Postel82] Jonathan B. Postel, *Simple Mail Transfer Protocol*, Request For Comments 821, Marina del Rey, CA: Information Sciences Institute, 1982.

[Reid91] Brian Reid, *Usenet Readership Summary Report for May 91*, Usenet message <1991Jun2.141124.12753@pa.dec.com> in *news.lists*, June, 1991.

[Torek91] Chris Torek, *Hash function for text in C*, Usenet message <27038@mimsy.umd.edu> in *comp.lang.c*, October, 1990.

Author Information

Rich Salz is a Senior Software Engineer at the Open Software Foundation, where is a member of the DCE group. His current areas of concentration are RPC and the distributed time service. He joined OSF after working at BBN for nearly five years, working on the Cronus Distributed Programming Environment. Rich attended MIT. He can be reached via U.S. Mail at Open Software Foundation; 11 Cambridge Center; Cambridge, MA 02142. Reach him electronically at rsalz@osf.org.

Tiled Virtual Memory for UNIX

James Franklin – Kodak Electronic Printing Systems

ABSTRACT

Many computer applications require the manipulation of large data arrays. These applications can behave badly under a paged virtual memory (VM) system, due to poor memory access patterns. One solution to this problem is tiling, a technique in which the arrays are partitioned into sub-arrays that map one-to-one with VM pages. Software implementations of tiling have been described in the literature, but none provide the speed and application transparency of a hardware solution.

We have implemented a hardware based, tiled VM within a version of the UNIX operating system. Based on a novel memory management unit and supporting kernel software, this tiled VM has proven to be an efficient environment for manipulating 2-dimensional arrays of data.

In this paper we discuss the kernel changes required to implement our tiled VM. We then compare tiled and paged versions of our VM system, and show that tiling results in a 50-fold reduction in working set size for a common class of image processing algorithms.

Introduction

There are many computer applications that require the manipulation of large, multi-dimensional arrays. Example applications include image processing, graphics rendering, numerical analysis involving large matrices, and simulation of physical systems. These applications can behave poorly under a conventional VM system, due to memory access patterns that result in large working sets[1] [6][4][1][9].

Applications that manipulate large arrays can benefit greatly from *tiling*, a technique in which large data arrays are partitioned into a number of identically sized sub-arrays, and the sub-arrays mapped (via software or hardware) to the underlying virtual pages. Algorithms for manipulating these tiled arrays often show dramatic reductions in working set size, resulting in less paging activity and faster execution times.

In a pioneering paper, McKellar and Coffman [6] investigate the performance of various matrix algorithms in a paging environment. They conclude that the use of sub-arrays can improve paging performance by orders of magnitude. Blinn [1] discusses the advantage of tiling in a graphics rendering application, and reports a 10-fold reduction in the number of page faults. Wada [9] presents a software implementation of tiling for image processing, and compares various tile replacement and prefetch algorithms. Another software implementation of tiling for image processing is described by Ryman [7]. But none of these implementations take advantage of the speed and application transparency that a hardware solution offers.

We have implemented a hardware based, tiled virtual memory within SUNOS, Sun Microsystems' implementation of the UNIX operating system. This tiled VM is based on a custom memory management unit called the IMMU and supporting kernel software. Together, they provide a tiled, shareable, virtual memory that we call *image memory*. As the name implies, we have used this tiled VM for image processing applications, although it would be equally useful for any of the applications mentioned above.

The IMMU and the tiled VM system described in this paper are currently being used by customers in a mid-range, color electronic prepress system called the Kodak Prophecy Color Publishing System.

In the next section we examine the motivation for using tiling techniques for large arrays. This is followed by an overview of the IMMU and the virtual to physical address translation in *IMMU Hardware*. In *Image Memory Concepts* we describe our implementation of image memory and the new system calls that support it. Tile fault handling, tile-in and tile-out, and new kernel daemons are described in *Kernel Software*. This is followed by a comparison of a tiled and paged version of our VM system in *VM Performance*. Finally, we close with a discussion of future work.

Tiling versus Paging

The fundamental benefit of tiling over paging is that processes that manipulate tiled arrays often show dramatic reductions in working set size. To illustrate this effect, we consider the manipulation of large images. Figure 1 shows a row-ordered, 2-dimensional array representing an image. The array has 3K rows containing 4K pixels per row, with each pixel occupying one byte. We want to manipulate this array on a machine that has 8 megabytes of physical memory available for paging, and a virtual page size of 4 kilobytes.

Now, consider a process that manipulates the array in paged virtual memory. In process virtual space, the array lies in a contiguous range of virtual addresses. Assuming that the first pixel is page aligned, each row fills one 4 kilobyte page (Figure

[1]Denning [2] defines the working set of a process to be the set of pages referenced by that process in some time interval of interest.

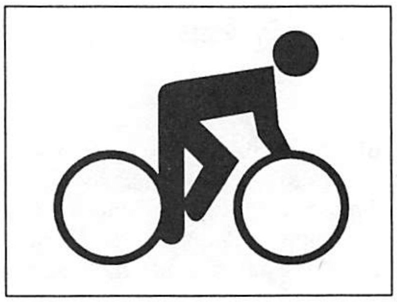


Figure 1: Row-Ordered Array in Virtual Memory

2). If the process accesses the array in row order, the array will be paged in and out of memory in an efficient fashion. But if the algorithm requires the process to access the array by columns, the process will suffer a page fault for every pixel accessed[2].

On the other hand, suppose the same process manipulates the same array in tiled virtual memory, using 256x256 byte tiles. In process virtual space, the array still lies in a contiguous range of virtual addresses. However, that range of addresses is now mapped to virtual tiles (Figure 3). If V is the virtual address of the first byte in the array, and V is mapped to physical tile T, then the virtual bytes at V, $V+1$, $\cdots$, $V+255$ are mapped to sequential physical bytes at T, $T+1$, $\cdots$, $T+255$. The virtual byte at $V+256$ is mapped to a different physical tile T'. The virtual byte at $V+4096$ (the first byte of the second array row) is again mapped to physical tile T, at $T+256$.

With tiled virtual memory, row and column access are equally efficient. If the array is accessed by columns, e.g., the process will suffer a tile fault every 256 pixels in the first column, and then run without faults for the next 255 columns.

Note the difference in working set size during column access to the paged and tiled arrays. For a column of the paged array, the working set size is 3K pages (the entire array), or 12 megabytes. For a column of the tiled array, the working set size is just 12 tiles or 786 kilobytes.

Tiled virtual memory can also reduce the working set size for many localized array operations. For example, consider an application that performs a local editing operation on the bicycle rider's shirt, such as a color or texture change. In paged memory (Figure 4), the working set size is approximately 1024 pages or 4 megabytes. In tiled memory (Figure 5), the working set size is 19 tiles or 1.2 megabytes.

[2]A column of pixels touches every virtual page in the array, requiring 12 megabytes of physical memory per column. The host machine has only 8 megabytes of physical memory available. Assuming a conventional LRU (Least Recently Used) replacement algorithm, linear passes through the array columns will cause the entire array to be repeatedly paged in and out of physical memory.

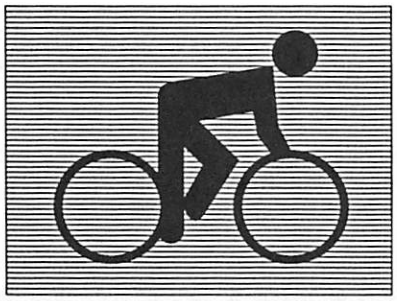


Figure 2: Mapping of Array to Virtual Pages

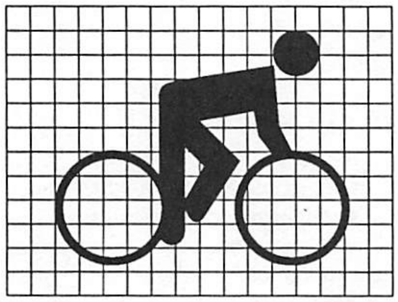


Figure 3: Mapping of Array to Virtual Tiles

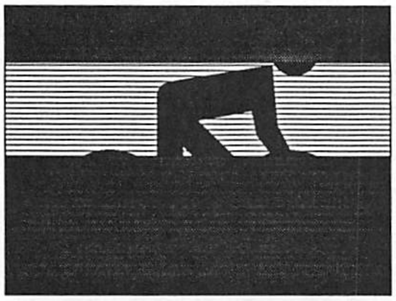


Figure 4: Pages Needed for Local Editing

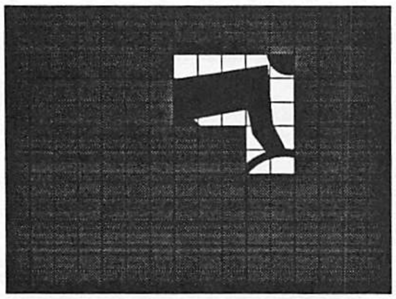


Figure 5: Tiles Needed for Local Editing

IMMU Hardware

Functional Overview

A functional overview of the IMMU hardware is shown in Figure 6. The IMMU consists of tile address mapping logic and a conventional MMU. Array virtual addresses from the CPU pass through the tile address mapping logic and are converted to tiled virtual addresses. These addresses then pass through the MMU and are converted to physical memory addresses via conventional means.

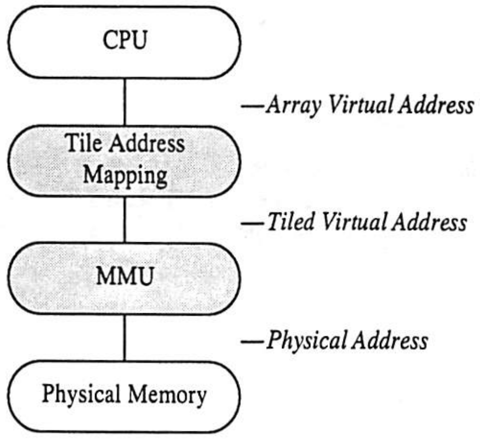


Figure 6: IMMU Functional Overview

The IMMU is located on the system bus, and responds to two address ranges: a 512 megabyte physical space, and a 1 gigabyte virtual space. All accesses to IMMU physical space are mapped directly to IMMU physical memory, which provides physical tiles to support the tiled virtual memory. Access to this physical space is restricted to supervisor mode, and is only done during tile-in, tile-out, and copy-on-write tile faults.

Access to IMMU virtual space invokes the IMMU virtual to physical mapping, and the access is redirected to the appropriate physical tile. During an access to IMMU virtual space, a tile fault will occur if the corresponding tile table entry is marked as invalid or if the access violates the virtual tile's protection. A tile fault causes the IMMU to generate a bus error signal.

Tile Address Mapping

When IMMU virtual space is accessed, the IMMU must translate the array virtual address to a tiled virtual address, which is then passed to the MMU. Internal views of an array virtual address and a tiled virtual address are shown in Figure 7 and Figure 8, respectively. An array virtual address consists of a Y address and an X address portion. The length of the Y and X portions can vary from 8 to 16 bits, corresponding to image dimensions from 256 bytes to 64 kilobytes. During tile address mapping, the lower 8 bits of the Y and X addresses (Y_{LOWER} and X_{LOWER}) are extracted and combined to form a tile offset. The upper portions of the Y and X addresses (Y_{UPPER} and X_{UPPER}) are extracted and combined to form a virtual tile number. Finally, the tile number and tile offset are concatenated to form the tiled virtual address.

In order to extract Y_{LOWER} and X_{UPPER} from an array virtual address, the IMMU needs to know the array's XSIZE – the number of bits in the X address. During tile address mapping, the upper bits of the array virtual address are used to index into an XSIZE table to obtain the number of bits in the X address. This effectively breaks the IMMU virtual space into 256 segments of 4 megabytes each. All images in the same segment have the same XSIZE. Images longer than 4 megabytes occupy multiple segments.

Y Address		X Address	
Y_{UPPER}	Y_{LOWER}	X_{UPPER}	X_{LOWER}

Figure 7: Anatomy of an Array Virtual Address

Tile Number		Tile Offset	
Y_{UPPER}	X_{UPPER}	Y_{LOWER}	X_{LOWER}

Figure 8: Anatomy of a Tiled Virtual Address

Image Memory Concepts

Image Memory as Shared Virtual Memory

Image memory is treated as a type of shared virtual memory. Processes request the allocation of an image in image memory and specify its attributes, such as size, read/write protection, and access rights. The kernel returns to the requesting process a unique image identifier for the allocated image. This identifier is used to reference the image or to allow another process to share the same image, by passing the image identifier. A reference count is maintained for each allocated image, and an image is freed when the reference count drops to zero. The separate components of a color picture (such as (u,v,L), (R,G,B), or matte) are stored as separate images in image memory.

Image memory is tiled in 64 kilobyte chunks to and from the swap device. Swap space for an image is allocated when an image is allocated, and deallocated when the image is deallocated. This swap space is associated with the allocated images, not with the processes that created them. Since image swap space is associated with each image, processes sharing an image therefore share the same image swap space.

New System Calls

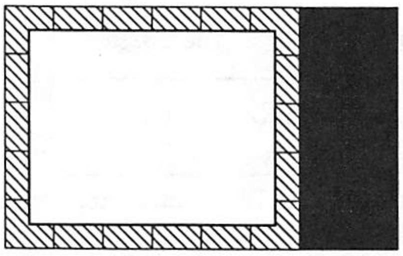
There are four new system calls to support tiled image memory. Image memory is allocated with *imem_alloc*, shared with *imem_share*, and freed with *imem_free*. The protection of an image can be changed with *imem_prot*.

imem_alloc

```
caddr_t
imem_alloc(x_dim, y_dim, image_area,
           access, fill_color)
int      x_dim, y_dim;
rect_t   *image_area;
access_e access;
int      fill_color;

typedef struct {
     int x_min, y_min, x_max,
         y_max;
} rect_t;

typedef enum {
        IMEM_PRIVATE, IMEM_SHARE
} access_e;
```



□ — Data pixels in *image_area* rectangle

▨ — Unused pixels in partially used tiles

■ — Unused pixels in read-only border tiles

Figure 9: An Allocated Image

imem_alloc allocates a section of image memory large enough to hold an image of size *x_dim* by *y_dim*. The *image_area* rectangle defines the area of the image that the user process will be writing to. By selecting appropriate values for *image_area* and (*x_dim*, *y_dim*), it is possible to obtain a constant color border around the working image area. The image is private to the current process if access is IMEM_PRIVATE, and shareable between processes if access is IMEM_SHARE. If *fill_color* is in the range 0-255 then all pixels in the allocated image are set to *fill_color*. If *fill_color* is NO_COLOR then all pixels in the allocated image are left unchanged, and will have undefined values.

Due to hardware constraints imposed by the IMMU, the specified *x_dim* value is rounded up to the next power of 2, and *y_dim* is rounded up to the next multiple of 256 as image memory is allocated. As a result, there can be border tiles around the image that do not overlap with the *image_area* rectangle. Any such tiles have their protection set to read-only, no-fault on write (see Figure 9). Any tiles that overlap the *image_area* rectangle have their protection set to read-write. *imem_prot* can be used to change this protection.

imem_share

```
bool_t
imem_share(image)
caddr_t image;
```

imem_share requests that the current process be allowed to share the previous allocated *image*. The image must have been specified as shareable by the process that allocated it.

imem_free

```
bool_t
imem_free(image)
caddr_t image
```

imem_free informs the kernel that the specified *image* is no longer needed by the current process. If no other process is currently sharing the image, it will be deleted.

imem_prot

```
bool_t
imem_prot(image, image_area, prot)
caddr_t image;
rect_t  *image_area;
prot_e  prot;

typedef enum {
        IMEM_PROT_RO,
        IMEM_PROT_RONF,
        IMEM_PROT_RW
} prot_e;
bool_t
```

imem_prot changes the protection of the previously allocated *image*. The new protection can be read-only, read-only with no fault on write, or read-write. The image must have been allocated by the current process or have been specified as shareable by the process that allocated it. The protection of all tiles in the region of the image specified by *image_area* is altered according to the value of *prot*. Protection can only be altered on entire tiles, not individual pixels.

Constant Color Tiles

Multiple virtual tiles are often mapped to the same physical tile. Tile sharing is desirable because it saves physical tile memory, and allows an allocated image to be quickly set to a default color, such as "paper white".

When an image is allocated using *imem_alloc*, a fill color is specified for the image. All tiles in the allocated image are *effectively* filled with the specified color. This is done by mapping all virtual tiles in the image to the same physical tile. The physical tile is filled with the specified color. Any border tiles around the image are permanently marked as read-only, no-fault on write. Any virtual tiles that overlap the *image_area* rectangle are marked as read-only and copy-on-write (see Figure 9).

Thus, a read of any pixel in the newly allocated image returns the fill color. A write to any pixel in a border tile is ignored, since all such tiles are marked no-fault on write. A write to any pixel in the *image_area* rectangle causes a tile-fault, since all such tiles are marked read-only.

The tile-fault handler detects that the virtual tile is marked copy-on-write, allocates a new physical tile, fills it with the color found in the read-only tile, re-maps the virtual tile to the new physical tile, and resets the protection bits for the virtual tile according to the original value of *prot* from imem_alloc. On return from the tile-fault handler the written-to pixel is updated in the newly mapped physical tile. Subsequent accesses to any pixels in the same virtual tile are directed to the newly mapped physical tile.

Kernel Software

Tile Faults

Tile faults can be generated by a process running on the host processor or by a hardware accelerator on the system bus. Faults generated by a process on the host processor cause a bus error to be received by the host processor, and are handled in a conventional fashion. Faults generated by a hardware accelerator cause a bus error to be received by the hardware accelerator, which generates a vectored interrupt to inform its device driver. These faults are handled by the tile-in daemon, which is discussed below.

Tile Swap I/O

Tile-in and tile-out are handled separately from SUNOS page-in and page-out. Under SUNOS versions 4.0 and 4.1, allocation of swap blocks is delayed until a page-out is required. Swap blocks can therefore appear in random positions on the swap device. In addition, swap blocks are equal in size to the virtual page size, which is 4 or 8 kilobytes.

To obtain maximum tile swap performance, we swap individual tiles to or from contiguous 64 kilobyte portions of the swap device. In addition, the current implementation allocates tile swap space when image memory is allocated, so that the entire image uses a contiguous portion of the swap device[3]. Both of these techniques minimize disk seek time during tile-in and tile-out. Tile i/o is done using a greatly simplified version of *physio* — the block i/o service routine of standard SUNOS. The resulting tile i/o subsystem is very efficient, and achieves approximately 95 percent of the theoretical disk bandwidth during operation.

Tile Daemons

Two new kernel processes are run to support the tiled virtual memory system. The first process, the *tile-out daemon*, tries to maintain an adequate supply of physical tiles in the free list to satisfy future tile faults. This process is similar to the page daemon process, and uses a conventional high/low water mark approach. Tiles that have been scavenged by the tile-out daemon are placed on a FIFO free list, and the virtual to physical mappings are maintained in a *tile cache*. If there is a tile fault on a tile that has recently been scavenged by the tile-out daemon, the physical tile is simply removed from the free list and the virtual to physical mapping restored.

The second process, the *tile-in daemon*, handles tile faults incurred by hardware accelerators. The hardware accelerators perform various image-processing operations on image memory, and are designed to be restartable. If one of the accelerators incurs a tile fault while processing an image, it generates an interrupt and halts. The interrupt causes the appropriate device driver interrupt routine to be invoked.

The interrupt routine gets the virtual tile address of the fault from the accelerator, and passes it to the tile-in daemon along with a notify routine. The interrupt routine then exits, leaving the accelerator halted. The tile-in daemon resolves the tile fault on behalf of the device driver[4], and then calls the notify routine. The notify routine then touches the restart bit on the accelerator, and the accelerator resumes its processing at the previously faulted address. The entire process takes just 200 microseconds on a SUN 4/330.

If this technique was not used, all the virtual tiles needed by an accelerator would have to be mapped and locked to physical tiles before the accelerator was started, similar to the way that pages are locked in prior to a disk transfer. But this is infeasible, given the potential size of the images and the lack of any knowledge of what tiles an accelerator is going to access. It would even be necessary to know which pixels were to be read, and which were

[3]In principle, this approach could cause problems with fragmentation of the image swap space, but this has not proven to be a problem in practice.

[4]The tile fault must be resolved by another process, since a driver can not sleep at interrupt level.

to be written, so that copy-on-write tiles could be properly mapped and copied.

Tile Replacement Algorithm

The choice of replacement algorithms for a tiled VM system is problematical, just as with a paged VM system [2][5][9]. The current implementation of the tile-out daemon uses a modified LRU (Least Recently Used) replacement algorithm. Unfortunately, LRU algorithms can perform poorly in common array processing operations [4][7]. If an array is accessed in a linear fashion in paged or tiled virtual memory, an LRU replacement algorithm acts just like a FIFO algorithm. If the array is larger than physical memory, successive passes through the array will cause the entire array to be copied to and from the swap device.

Despite the potential problems with LRU replacement, we have found it to provide acceptable VM performance in practice. The tile cache (discussed above) has proven to be very helpful in improving system performance.

We have experimented with a RANDOM replacement algorithm, and found it to be much slower than the LRU algorithm, even in cases where the LRU algorithm performs poorly. The dismal performance of the RANDOM replacement algorithm appears to be caused by the complete failure of the tile cache — with random tiles in the free list, the tile cache hit rate falls close to zero.

We are currently investigating the use of LIFO (Last-In, First-Out) replacement. If large data arrays are accessed repeatedly, in a sequential fashion, LIFO replacement may be much more effective than LRU replacement. Initial tests have shown that LIFO replacement reduces tile i/o by 25 to 50 percent in our application.

To illustrate this effect, suppose that we have a 20 megabyte array, and that the operating system has 16 megabytes of physical memory available for paging. Now, suppose that the array has already been accessed sequentially at least once. With LRU replacement the last 16 megabytes of the array remain in memory, and the first 4 megabytes are swapped out. With LIFO replacement the valid and swapped sections are reversed, i.e., the first 16 megabytes of the array are valid, and the last 4 megabytes are swapped out.

When the array is processed a second time, using LRU replacement, the initial portion of the array is invalid, and must be swapped in from disk. But for each page swapped in, another page must be swapped out. As a result, the second sequential pass through the array causes every page in the array to be swapped in once and swapped out once, for a total of 40 megabytes of page i/o.

But when the array is processed a second time, using LIFO replacement, the initial 16 megabytes of the array are still valid. No page faults occur until the last 4 megabytes of the array are accessed. Just as with LRU replacement, for each of these pages swapped in, another page must be swapped out. So with LIFO replacement, a total of 8 megabytes of page i/o are required.

In general, for repeated sequential access to an array of N bytes on an operating system with P bytes of physical memory, with $N>P$, LRU replacement will require $2N$ bytes of page i/o, and LIFO replacement will require $2(N-P)$ bytes of page i/o.

We expect to continue investigating replacement algorithms in the future, including LIFO and the most-distant algorithm discussed in Future Work.

VM Performance

By making a trivial change to the kernel code for *imem_alloc*[5], we are able to turn our tiled VM into a conventional paged VM. The tiled version uses 64 kilobyte tiles, the paged version uses 64 kilobyte pages. Every other aspect of the two kernels is identical, including use of the IMMU hardware, fault handling, swap space allocation, swap i/o transfer size, replacement algorithm, and tile/page daemons.

These otherwise identical VM systems are ideal for comparing the performance of tiling and paging on a common image processing operation such as filtering[6]. To do the comparison, we use one of our hardware accelerators to filter a 12 megabyte (4000x3000 pixel) greyscale image. The filtering operation requires a horizontal and a vertical pass, so the entire image is read and written twice: once by rows; once by columns.

Our test filters the image once for each of a range of physical memory sizes, recording the execution time for each. We repeat the test using both the tiled and the paged VM system. At the start of the test, 16 megabytes of physical memory are available to the VM system for image storage. On each pass of the test the following steps are performed:

1. The required images are allocated, initialized, and forced out of physical memory;
2. The filtering operation is performed and the processing time is recorded;
3. The images are deallocated; and

[5]We simply set XSIZE (the number of bits in the X address portion of the virtual address) to 8. This effectively transforms a request for an *N by M* tile image into a request for an *(N x M) by* 1 tile image. That is, the allocated image is 1 tile wide, regardless of the requested width.

[6]Filtering is used for a variety of image processing operations, including rotation, scaling, sharpening, blurring, and warping.

4. The amount of physical memory is reduced by 256 kilobytes.

The passes continue until the VM system starts thrashing.

Figure 10 shows the results of these tests as a graph of execution time vs. physical memory size. The paged VM system starts thrashing when physical memory drops to 13 megabytes[7]. The tiled VM system, on the other hand, exhibits a linear degradation in performance until physical memory drops to 2 megabytes. Below 2 megabytes there is a steeper but still roughly linear degradation.

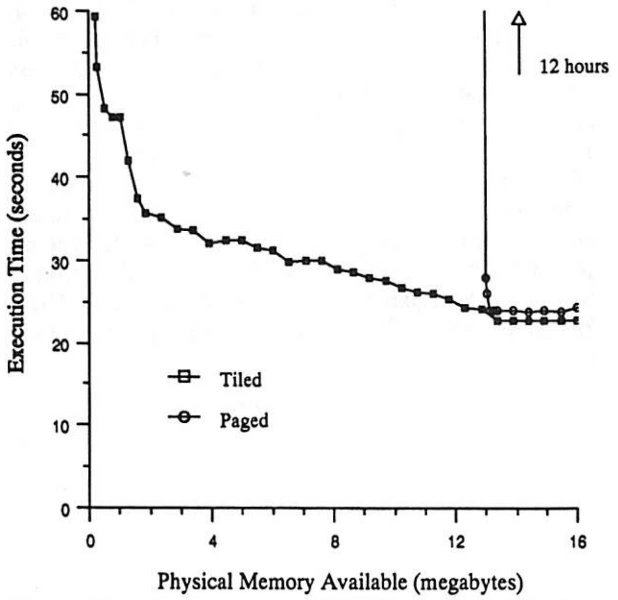


Figure 10: Performance of VM System on Filtering Task

For this filtering task, the paged VM system requires at least 13 megabytes of physical memory. The tiled VM system continues to function with 1/4 megabyte of memory and only a doubling in the nominal filtering time. This is a 50-fold reduction in working set size.

Future Work

Tile Replacement Algorithms

As discussed in the *Kernel Software* section, the tile-out daemon uses a modified LRU algorithm, and this algorithm can perform poorly for some common array access patterns.

Wada [9] suggests the use of a *most-distant* replacement algorithm for image processing applications. In this algorithm, the tiles most distant from the last accessed tile are candidates for replacement. This algorithm appears to be well behaved under many of the common access patterns in image processing. However, it is more computationally expensive than other common replacement algorithms, and also requires knowledge of the last tile accessed. This information is not available from our IMMU hardware.

We would like to investigate whether an approximation to the most-distant algorithm would be effective. One possible approach would be to use a conventional clock algorithm to clear all of the tile reference bits at regular intervals. When the tile-out daemon runs, the centroid of the recently referenced tiles could be used as an approximation to the most recently used tile.

Tile Prefetch Algorithms

The current tiled VM system is *demand tiled* — a tile is fetched from the swap device only when a tile fault occurs. It may be desirable to modify the tile fault handler to detect common image access patterns (such as row or column access), and then asynchronously prefetch entire rows or columns of tiles before they are needed.

Alternatively, it may be easier to add a system call that would allow the application process to provide hints to the kernel about intended image access. The hints should include the region of the image to be processed and the image access pattern.

We would also like to investigate the possibility of coupling images with respect to tile faults, so that a tile fault on one color component of a picture would cause an implied fault on the corresponding tiles in the other color components. Similar coupling of source and destination images may also be useful.

Extensibility to Higher Dimensions

The current implementation of the IMMU hardware and software provides an efficient environment for processing large 2-dimensional arrays. However, the array virtual to tiled virtual address translation (as performed by the IMMU) is easily extensible to higher dimensions. The only fundamental constraint is the size of a virtual address on the host processor.

Most modern processors provide 32-bit virtual addresses. In the current implementation, both the X and Y portions of an array virtual address can be up to 16 bits. A tiling architecture to handle 3-D or 4-D arrays would require reductions in the maximum dimensions of the arrays, but would still be useful for some applications. However, future trends are definitely toward larger virtual addresses [8], and 64-bit machines are already becoming available. On such a machine, a tiling architecture for 3-D or 4-D arrays would be quite interesting.

Integrated MMU and IMMU

The IMMU is completely separate from the host processor's MMU and is situated on the system bus. This limits the potential performance of the image memory system, due to the overhead in

[7]This is the size of the image plus free space for the VM system to use for tile fault resolution.

addressing the system bus and the inability to take advantage of the cache hierarchy on the host processor.

The IMMU is very similar to a conventional MMU, with the exception of the array virtual address to tiled virtual address mapping. Integration of this mapping logic into a conventional MMU should be straightforward, but is beyond our means.

Conclusions

We have implemented a tiled virtual memory for UNIX, based on a custom MMU and supporting kernel software. Together, they provide a tiled, shareable, virtual memory that has proven to be an efficient environment for manipulating 2-dimensional arrays of data.

Use of the tiled VM presented here is nearly transparent to application software and programmers. In fact, the same executable binaries run on systems with and without tiled VM, the only difference showing up in array handling performance. For arrays that fit in memory, tiled and paged VM are equal in performance. For arrays larger than physical memory, tiled VM provides much higher performance.

This tiled virtual memory has been used successfully in a commercial product for the color electronic prepress industry. Although we have used this tiled VM solely for image processing applications, we believe it would be equally useful in a variety of other applications.

Acknowledgements

Gary Newman was responsible for the conceptual design of the IMMU and its use in a tiled virtual memory system, and later guided the development of both the IMMU and the kernel software described here. Steve McLafferty and Bob Getz implemented the IMMU hardware.

References

[1] Blinn, J. F., "The Truth About Texture Mapping", IEEE Computer Graphics and Applications (March 1990), 78-83.

[2] Denning, P. J., "The Working Set Model for Program Behavior", Commun. ACM 11(5) (May 1968), 323-333.

[3] Denning, P. J., "Virtual Memory", ACM Comput. Surv., 2(3) (Sept 1970), 153-189.

[4] Hatfield, D. J., and Gerald, J., "Program Restructuring for Virtual Memory", IBM Systems Journal 10(3) (March 1971), 168-192.

[5] Madnick, S. E., and Donovan, J. J., *Operating Systems*, McGraw-Hill, New York, 1974.

[6] McKellar, A. C., and Coffman, E. G., "Organizing Matrices and Matrix Operations for Paged Memory Systems", Commun. ACM 12(3) (March 1969), 153-165.

[7] Ryman, A., "Personal Systems Image Application Architecture: Lessons Learned from the ImagEdit Program", IBM Systems Journal 29(3) (Sept 1990), 408-420.

[8] Slater, M., "Is 64 bits the next step?", Microprocessor Report, 5(4) (March 1991), 3-4.

[9] Wada, B. T., "A Virtual Memory System for Picture Processing", Commun. ACM 27(5) (May 1984), 444-454.

Author Information

Jim Franklin received a BA in Mathematics from Cornell University in 1974, and an MS in Computer Science from the University of Maryland, College Park in 1976. He was a member of technical staff at Bell Laboratories until 1980, working on software tools, and then joined Automatix, working on programming languages for robotic systems. In 1986 he joined Kodak Electronic Printing Systems, where he works as a consulting engineer in the Color Image Products Group, investigating operating system issues for new imaging platforms. He can be reached via U.S. Mail at Kodak Electronic Printing Systems; 164 Lexington Road; Billerica, MA 01821. His electronic mail address is jwf@keps.kodak.com or uunet!keps.kodak.com!jwf .

A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines

Ramesh Balan, Kurt Gollhardt – UNIX System Laboratories

ABSTRACT

This paper describes the design and implementation of the UNIX® SVR4.2 Virtual Memory (VM) Hardware Address Translation (HAT) layer that can be used as a model for other multiprocessor (MP) platforms in terms of scalability and MP related interfaces between the HAT layer and the machine independent layer. SVR4.2 is a SVR4.1 ES based kernel that supports shared memory multiprocessors and light weight processes in a shared address space. By implementing a fine-grained locking mechanism, a lazy Translation Lookaside Buffer (TLB) shootdown evaluation policy and other improvements over the SVR4 design, the memory management feature is made scalable in terms of number of processors as well as size of memory supported. Providing a small set of interfaces between the machine dependent and independent layers for TLB consistency and a simple set of locking requirements between the two layers, SVR4.2 facilitates the portability of the memory management feature to other multiprocessor platforms.

Introduction

A scalable and portable HAT layer that supports multiprocessors and multiple threads in an address space is described in this paper. The scalability of the implementation is primarily due to three reasons:

- The TLB shootdown policy and algorithms.
- A fine-grained locking scheme that allows memory management as a whole to be scalable with respect to number of processors.
- Design to support large physical memory configurations.

A small set of well defined MP related HAT interfaces is introduced for use by other layers of the kernel. The purpose of these HAT functions is to maintain TLB consistency in a multiprocessor environment. SVR4.2 does not assume hardware support for TLB consistency[1] and the support is provided by the HAT layer.

The HAT layer is the Memory Mangement Unit (MMU) dependent part of the memory management facility in SVR4.0 UNIX implementations. Other UNIX Virtual Memory implementations also usually contain such a machine dependent layer. In SVR4.2 (a derivative of SVR4.1 that provides support for multiprocessors and light weight processes), all but a small portion of rest of the VM subsystem is machine independent.

Traditionally, most of the porting effort is spent on implementing the HAT layer when porting SVR4 memory management feature to various architectures. This effort is much more complex in a multiprocessor environment. Also, typically, scalability issues are not emphasized during porting efforts. By providing a well defined set of interfaces and a simple locking protocol, the porting effort will be routine without any loss in the performance of the system.

Related Work

Previous work done on providing a general interface for the hardware dependent layer of VM includes the MACH pmap layer [1] and the original SVR4 HAT layer interfaces that were derived from SunOS [2]. The TLB shootdown policy implemented in SVR4.2 is similar to the MACH policy [3], however kernel address space shootdowns are handled differently from the user address space. Several solutions to TLB consistency with and without assuming hardware cache consistency have been discussed in various papers [4]. An implementation of TLB synchronization that uses a particular TLB format (TLB ID entry) has been described in [5]. The SVR4.2 implementation does not expect the TLB to contain any fields such as TLB ID other than a subset of fields in the page table entry. However, there are two areas in which SVR4.2 implementation of TLB shootdowns is machine dependent: one is when clearing the page table entries of other processors which is dependent on the MMU structure and the other is in sending inter-processor interrupts for synchronization of the processors whose TLB is being shot down.

[1]However, cache coherence is assumed to be supported by the hardware.

Background

The SVR4.0 HAT data structures were retained for SVR4.2. The reason for this is that the data structures efficiently support large, sparse address spaces in terms of space and time. The principal factor behind this efficiency is the *mapping chunks* data structure. A *mapping chunk* is used to keep track of all virtual mappings to a physical page. Each page table entry has a corresponding mapping chunk entry and each physical page has a linked list of mapping chunk entries that denotes the virtual translations to the page (*mapping chain*). The size of a mapping chunk is much smaller than that of a page table[2]. Non-active translations does not have an entry in the mapping chunk. Due to this reason, sparsely populated page tables waste very little space for providing the *mapping chain*. When operating on a large address range[3], all the page table chunks that does not have a corresponding mapping chunk are skipped, and no time is spent looking at the non existent page table entries.

The Uniprocessor (UP) interfaces from the SVR4.0 HAT layer have also been retained since the interfaces have been found to be sufficient in supporting different architectures that SVR4 has been ported to so far (including Intel386, SPARC®, Motorola 88000, MIPS). The most frequently executed UP HAT functionalities in SVR4 were to load a translation to a given page (*hat_memload()*), to unload translations for a range of addresses (*hat_unload()*) and to unload all translations to a given physical page (*hat_pageunload()*).

The reference port for SVR4.2 is on an Intel386/486 architecture and thus the initial HAT implementation is targeted for the Intel386 MMU. The following is a list of its features that are of interest:

- The Intel386 MMU uses a two level page table structure to define an address space [6]. When references to the page table entries are denoted as level 1 entries or level 2 entries in the sections below, they are in regards to this structure.
- Level 1 is the page table directory consisting of 1024 entries, each of which points to a page table. This page table is referred to as the level 2 page table.
- Level 2 page table consists of 1024 entries, each of which point to a physical page.
- The physical page size is 4096 bytes.
- The modify and reference bits are in the page table entry and are updated by the hardware.
- The i386 also provides an interlocking facility when accessing the reference and modify bits; i.e. no other accesses to the page table entry are possible when the hardware is changing these bits for that entry.

The i386 architecture can support 4 Gigabytes of virtual address space. In many RISC architectures (such as MIPS®), the modify and reference bits are simulated in software and thus, unlike the Intel386 implementation, TLB shootdowns are not required when these bits are modified.

Multiprocessor Interfaces

Most MMUs implement a simple cache known as Translation Lookaside Buffer for caching virtual to physical translations to avoid real memory accesses. In a multiprocessor environment the same virtual address can reside in multiple TLBs and the coherence of these translations needs to be maintained between the TLBs. In SVR4.2, all the exported MP related HAT interfaces are used for maintaining the TLB consistency. The number of active CPUs in the system for the kernel address space and the number of CPUs a user address space (execution entity : a *process* (consisting of one or more *light weight processes*)) is associated with is recorded to do selective TLB flushes. This is referred to as TLB accounting. The HAT layer records the TLB accounting in a HAT data structure that is associated with each address space, including the kernel address space.

All online CPUs in the system can execute in the context of the kernel address space (*kas*). The kernel address space HAT accounting structure records the current set of online CPUs. Two HAT functions are provided for establishing this accounting when bringing CPUs online or offline. These functions are used in accounting which processors' TLBs will be flushed for the kernel address space.

- `hat_online()`: Called when onlining an engine (CPU) in the system. Sets *active cpu count* field and the processor's bit in the *kas* HAT structure. It also flushes the engine's TLB.
- `hat_offline()`: Called when taking an engine (CPU) offline in the system. Clears the processor's bit set in `hat_online()` and decrements the count of active *cpus* in *kas* HAT structure.

The processor accounting for user level address spaces for shooting down TLBs is done at context switching time. Since threads within an address space can be running at the same time on different CPUs, the CPUs that are executing in the context of the same address space must be known to perform selective TLB flushes. The following interfaces are used when scheduling a light weight process (LWP) on any CPU in the system.

- `hat_asload(as)`: Called when context switching to a new LWP. It adds the

[2]In the i386 implementation, the size of a mapping chunk is 1/32nd of page table size.

[3]Such as unloading an address range or changing protections.

processor to the active engine (processor) accounting in the HAT structure of this address space *as* and loads this address space into the MMU (just the level 1 page table entries on the i386 architecture).

- `hat_asunload(as, flag)`: Called when context switching out a LWP. It unloads the MMU mappings for this process (again, just the level 1 translations on the i386) and takes the engine out of the active engine accounting of the HAT structure. The flag parameter indicates whether the caller wants a TLB flush to be done by this function after unloading the mappings[4]. Except for the CPU accounting, the rest of the functionality needs to be done on a UP platform as well[5]. Note that there is no need to call *hat_asunload()* if the context switch is to select a LWP in the same process.

The following are the HAT interfaces for lazy shootdown of TLBs used only on the kernel address space by the kernel segment drivers. To implement lazy TLB shootdowns (details of which is explained later), an object opaque to all other layers of VM except the HAT layer called a *cookie*, is maintained. The *cookie* reflects the age of virtual translations with respect to the TLB. In the i386 HAT implementation, the *cookie* is a timestamp but it could be a counter of some sort in other implementations. The state of the TLB the HAT records is the timestamp of the last TLB flush. The state of a virtual address will be explained in section 6.1.1. The following are the interfaces:

- `hat_getshootcookie()`: Returns an opaque value that indicates the "age" of a TLB, which is used for lazy shootdown.
- `hat_shootdown(cookie_t cookie, u_int flag)`: TLB shootdown routine for kernel address space. If any of the active CPUs in the system has an older *cookie* than the passed-in *cookie*, then the TLBs of these CPUs will be flushed. The *flag* argument is used by clients which do not use lazy shootdown[6], so all the CPUs in the system are flushed regardless of the *cookie* passed in.

[4]The SVR4.2 i386 context switch implementation does not request *hat_asunload()* to flush the TLB. This is because it has to flush the TLB after copying the page table entries for the U area of the new process it is loading. Thus it forgoes the TLB flush after unloading the mappings of the old process.

[5]The TLB flush is not necessary on some architectures whose MMUs (such as SPARC and MIPS) provide the context number as part of every TLB entry and on those architectures where TLBs are flushed on each context switch.

[6]There is no such client in SVR4.2.

The TLB shootdown interfaces for user level address spaces are not seen outside the HAT layer – the shootdown is immediate and it is done during one of the following HAT operations: unloading a translation, changing the protection of a page (only in the case of restricting permissions), remapping a virtual address to a different physical address, and in the case of clearing a modify bit of a page table entry (architecture specific).

Scalability Solutions

This section will discuss some of the features that makes the SVR4.2 implementation scalable.

TLB Shootdowns

On platforms that does not support TLB consistency in hardware, a multiprocessor kernel needs to maintain the consistency for translations that are cached in several processors' TLBs. The TLB is a common feature of present day architectures since it avoids any memory accesses (two in the case of a i386 architecture) in translating a virtual address to physical address if the address is present in the TLB cache. The shootdown algorithms depend on the existence of a hardware facility to issue cross-processor interrupts. TLBs are fully flushed[7] as opposed to flushing single TLB lines [5]. Since, the shootdown algorithms are MMU architecture dependent, they are part of the HAT layer in SVR4.2.

A lazy shootdown policy for the kernel address space has been used whereas immediate shootdowns are employed for the user address space. Since the kernel virtual address usage is in the control of the kernel, a lazy evaluation of the inconsistent TLB states can be done. However, for a user address space, multiple TLBs need to be immediately brought to a consistent state since SVR4.2 supports multiple LWPs in an address space which can concurrently execute on multiple CPUs.

Lazy Shootdowns

A lazy evaluation policy is very important for the kernel address space. When a kernel virtual address translation is unloaded, all processors' TLBs in the system need to be brought to a consistent state. This is because all processors in the system share the kernel address space (in a symmetric multiprocessor architecture). Delaying shootdowns may avoid doing the shootdowns entirely since the TLBs might be flushed already when the evaluation is done (due to a context switch, for example).

The kernel segment drivers essentially determine the laziness of a shootdown in kernel address space. Two major users of this policy in SVR4.2 are the *segkmem* driver which manages the permanently resident kernel memory and the *segmap* driver which

[7]The Intel386 architecture does not support single line TLB flushes except through the use of an unsupported multi-instruction sequence.

manages transient file mappings used by file system read and write system calls.

When a kernel virtual address is freed by a kernel thread, then typically that address would need to be flushed from all the TLBs in the system. But the SVR4.2 *segkmem* driver delays this shootdown until this address is about to be reused by the kernel. The virtual space managed by the *segkmem* driver is represented as a bitmap and the bitmap itself is divided into zones (the size of the zone is a tuneable; the default value is 16 bytes). Each zone has associated with it a *cookie* (explained in the previous section), which is set when an address in the zone is freed. At the time of allocation, when it is found that a page is allocated from a freed zone whose addresses have still not been flushed from the TLBs, *hat_shootdown()* is called with the *cookie* associated with the zone as an argument. What *hat_shootdown()* does with this *cookie* will be explained shortly.

Similarly, *segmap* manages its virtual space in fixed-size *chunks* (configured as 8K as the default value) and each *chunk* has an associated *cookie*. Unlike *segkmem*, however, when a *segmap chunk* is freed (last reference is released), the *cookie* for the chunk is set through the *hat_getshootcookie()* interface but the translations are not unloaded. Instead, this chunk is linked on to a list; the *segmap* aging daemon periodically looks at this list and unloads the translations at this time but does not perform a shootdown of the unloaded addresses. When the chunk is then reused by *segmap*, it calls *hat_shootdown* with the associated *cookie*. The shootdown can be delayed after the unloading since no other context can access this file page in the mean time. This technique allows us to eliminate the shootdown entirely, if the chunk is reused with the same identity (same physical pages) before it is aged.

Lazy Shootdown Algorithm

Inside the HAT layer, a *cookie* is associated with each processor that denotes when the processor's TLB was flushed last. In a separate global variable, the *cookie* of the least recently flushed TLB is maintained. If the *cookie* passed in to *hat_shootdown()* is older than this value, then it immediately returns since it knows that all the TLBs in the system have been flushed since the *cookie* was acquired. If this is not the case, the following steps are executed by the initiator (the context that is initiating the shootdown):

1. Acquires a global spin lock. This spin lock disallows the active processor set of the system from changing underneath. It also serializes lazy shootdowns in order to set the *cookie* for each processor.
2. Scans the list of all processors that have been *hat_online()*'ed (see interface definition in the last section) and selects all the processors whose *cookie* is "older" than the passed in *cookie*. While selecting the processors to be interrupted, it recomputes the least recently flushed value and sets the *cookie* for each processor it selects (to *lbolt* in our implementation).
3. Sends cross processor interrupts to the processors.
4. Unlocks the global spin lock it acquired earlier once all the responders have begun processing the interrupt.

The responders (processors at the receiving end of these interrupts) then flush their own TLBs before again becoming active. The cross processor interrupt executes at the highest interrupt priority level (ipl) in the system because no interrupts can be allowed while servicing a shootdown. Otherwise, this could result in a deadlock if the interrupt level routine causes a shootdown itself. This interrupt level is even higher than the normal "block-all" interrupts level (splhi) to avoid latency problems; we are careful to avoid changing anything in the cross-processor interrupt service routines which could interfere with splhi-protected critical regions. Note that the responders do not wait for any synchronization with the initiator in this algorithm. All they have to do is a TLB flush since the translations have been modified earlier by the segment drivers. The initiator does not wait for all the responders to complete their operation.

Immediate Shootdowns

The interfaces for immediate shootdowns employed for the user address space are hidden in the HAT layer and are not exported to other layers in VM. This is because immediate shootdowns are caused only by operations within the HAT layer such as unloading a translation and changing protections for a translation.

Immediate Shootdown Algorithm

The algorithm for immediate shootdown is similar to the lazy shootdown algorithm. The following steps are executed by the initiator:

1. Grab the same global spin lock that we acquire in the lazy algorithm for the same reason (to keep anybody else from changing the active processor set or performing another shootdown).
2. Send cross-processor interrupts to all the processors that share this address space (the processor list that is updated by *hat_asload* and *hat_asunload*). Unlike the lazy algorithm, the responders spin waiting on synchronization with the initiator.
3. Modify the page table entries (level 2 entries) as appropriate for the operation (zero page table entries if unloading translations, change the protection bit or clear the modifying bit if *sync*ing the page table entry to the page

structure).

4. Increment the counter that the responders are spinning on. The responders perform a TLB flush and return from the interrupt.
5. Perform a TLB flush for the initiator's processor.
6. Unlock the global spin lock acquired earlier. The initiator again – as in the lazy case – does not wait for the responders to finish flushing their TLBs.

This algorithm has been optimized for the i386 architecture when the initiator has to modify a large range of page table entries (example: when unloading a large range of addresses). The initiator holds the HAT resource lock (a spin lock) that is associated with the address space being modified at the outset of the algorithm. After the responders are in a spinning state, instead of changing all the page table entries the initiator just unloads the level 1 entries for the affected page tables. Thus, the initiator spends less time when all other processors are spinning. The initiator then increments the counter that releases the responders from spinning on the barrier. The responders then flush their TLB before returning from the interrupt. If any of the LWPs running on the responders try to access the inconsistent page table entry, it will take a fault because of the non-existent level 1 entry. The trap code will then try to acquire the HAT resource lock and will block until the initiator releases the HAT lock. This reduces the time processors spin uselessly in the shootdown algorithm.

Pageout

The implementation of local working set aging for pageout in SVR4.2 also prevents shootdowns when compared to the global pageout policy in SVR4. The global pageout daemon scans all the physical pages in the system and clears the modify bit if the bit is set for a page (after calling VOP_PUTPAGE() on the page) or clears the reference bit if it is set. Both of these actions would require shootdowns (since these bits are in the page table entry). But with the working set aging, the process to be *aged* is seized; i.e. all the LWPs in the process except the current context are brought to a quiescent state. Thus, there is no need to shootdown when modifying the page table entries. The i386 context switch code flushes the TLBs when switching back in these LWPs.

Other Architectures

The above mentioned interfaces and algorithms provide flexibility in supporting various architectures, requiring minimal changes to them.

Architectures supporting single TLB entry flushes:

- The lazy shootdown algorithm need not change at all. Even though the algorithm flushes the whole TLB, most shootdowns are totally avoided by this policy (see section "Performance Data") and thus result in very little overhead when compared to flushing individual entries.
- The immediate shootdown interfaces (both the initiator and the responder) would change to take in the address range as an argument and flush just those entries. Since these interfaces are not exported to other VM layers, changing the interfaces is acceptable.
- There may be a point in such architectures where flushing a whole TLB is cheaper if the number of lines to be flushed in the TLB is too large. The algorithms should take this into account when deciding which is more efficient.

Architectures whose TLB entries contain a field for context number:

- It is unnecessary to flush the TLB on context switches.
- The lazy shootdown algorithm would not change. If no other local TLB flushes are done by the kernel[8], all the *cookie*s associated with the processors would be in the same state and only one *cookie* would be needed.
- For the user address space, a lazy shootdown algorithm may be possible as implemented in [5].
- No changes to interfaces are necessary.

Locking Design

The locking design implemented for the VM subsystem as a whole should scale well on parallel activities (intra-process and inter-process) that occur on the system. The primary motive in arriving at the current locking model was to keep things simple and not to have the locking requirements between the VM layers (the page layer, the segment layer and the HAT layer) too complex. As a result, porting of this HAT layer to other architectures should be almost as straightforward as a Uniprocessor HAT layer.

The principal locks in the VM layer are:

- Page Layer
    - A global spin lock in the page layer for protecting the page hash chains
    - A per page spin lock for mutexing the fields of the page structure
    - A read/write sleep lock which is acquired in reader mode to ensure that the page state, identity and data are valid and remain so and acquired in writer mode if modifying any of the above.
- Segment Layer (user segment driver)
    - A reader/writer lock per segment. This lock is acquired in writer mode when changing the attributes of a segment

[8]This is not the case in SVR4.2.

(such as protection) and in reader mode when the attributes of the segment are to remain valid for the duration of operation.
  - A per segment spin lock which guards the sleep lock.
- HAT Layer
  - There is only one spin lock associated with each address space for guarding the HAT resources.

Making the HAT lock finer grained by moving it to the page table level was considered but decided it wouldn't be much of a gain for the following reasons: most UNIX processes fit in one page table and there would be extra locking round trips for HAT functions that cross page tables. If found necessary, other ports can move this to a page table level (architectures where the page table size is small) without any need to change the locking requirements.

Analysis of the Locking design

Two widely occurring system events in UNIX systems, page faults and fork()/exit() operation, would be a good indicator of scalability in the VM layer.

- When generating concurrent page faults in different address spaces, the only lock contention will be for the global page layer spin lock that is guarding the page hash chains. The lock hold time for this lock is very low. There will be different instances of the HAT lock (due to different address spaces) and segment locks (faulting on different segments).
- For concurrent page faults generated within a process among its LWPs, there would be contention for the HAT resource lock but the lock hold time during loading of a translation will again be very small. Faulting on the same segment by various LWPs would cause contention for the per segment sleep lock. Some faults require the lock to be held only in reader mode and thus allows for parallelism between such faults at the segment layer.
- When concurrent fork()/exit() operations take place in different address spaces, the only contention at the VM level would be for locks at the *anon* layer (which manages anonymous pages) and at the *swap* layer for reserving anon pages and swap space for the child processes respectively. Again, the lock hold times during the reservation operation would be very small.
- Intra-process concurrent fork()/exit() operations could cause lock contention at the HAT layer and the segment layer but both the locks will be held only while each segment is being copied. Reducing the lock hold time on the HAT resource lock by dropping the HAT lock after copying each *mapping chunk* (32 page table entries) is being considered.

Examples of the Locking requirements for HAT interfaces

To get an idea of the locking requirements, some of the HAT interfaces are listed here. In all these operations, the HAT resource lock is acquired by the HAT layer.

`hat_memload()`: Load a virtual address translation. Called with the reader/writer lock for the physical page held. The caller can not hold any spin locks.

`hat_unload()`: Unload a range of virtual address translations. The caller need not hold any spin locks. This routine acquires the spin lock associated with the physical page structure in order to modify the mapping chain for the page.

`hat_pageunload()`: Unload all the virtual translations to a given physical page. The spin lock for the page is held by the caller.

Physical Memory Scalability

SVR4 had a limit on the physical memory it was able to support on the i386 platform. Changes were made in SVR4.2 to avoid this limit[9]. Several kernel functions in SVR4 relied on the fact that all of physical memory in a machine is mapped into the kernel virtual space. These functions generally need to get a virtual address from a given physical address in a non-blocking fashion. On the Intel386 reference port, out of the available 4 Gigabyte virtual address space, the user address space was given 3 Gbytes and the kernel 1 Gbyte virtual space. The kernel virtual itself was divided at kernel boot time among different kernel segment drivers (*segkmem*, *segmap* and *segu* (which manages the simultaneous mapping of several processes' U areas at the same time)). After this division, only 256 Mbytes of physical memory could be mapped into the kernel virtual space. The default layouts of the kernel memory map could be changed to make this limit bigger but there would still be a limit. All the kernel functions that expect this non-blocking behaviour were modified in SVR4.2 to eliminate this restriction in one of the following two ways:

- Cache the needed virtual address.
- Create and destroy virtual mappings to a given physical page as needed.

The HAT layer was one of the primary users of this "magic mapping" in using it to get to the virtual address of a level 2 page table entry from the page frame number stored in the level 1 page table entry. It was changed to cache the virtual address in the HAT structure itself. Going into the details of all the changes in the kernel is beyond the scope of this paper.

[9]This decision was not influenced by any MP related issues.

Performance Data

Most features of SVR4.2 have been completed and performance measurements are beginning to be collected for the system. Thus the performance measurements presented here are by no means the optimal figures for SVR4.2.

Shootdown Measurements

Some measurements were taken on how well our shootdown algorithms (lazy and immediate) scale with respect to number of processors. Scalability of the basic cost of the shootdowns, the lazy shootdown algorithm and the immediate shootdowns algorithm were measured. The following measurement process was used in collecting the data:

- The measurements were taken on a Sequent Symmetry platform which has 6 Intel 386 processors at 20Mhz.
- Ten samples of each measurement were taken, and their mean value was used.
- A measurement tool called *casper* was used to measure the time spent in different windows of the kernel code paths in units of microseconds.
- All our measurements reflect the time spent by the initiator.

The time spent by the responders in the lazy algorithm would be a fixed time constant (time taken to flush its TLB). In the immediate shootdown case, the time spent by the responders would be upper bounded by the time spent by the initiator.

The basic cost of shootdowns was computed on the Symmetry by calling *hat_shootdown()* with HAT_NOCOOKIE as an argument, which shoots down all processors in the system without any other computation. Figure 1 is the graph that illustrates the results.

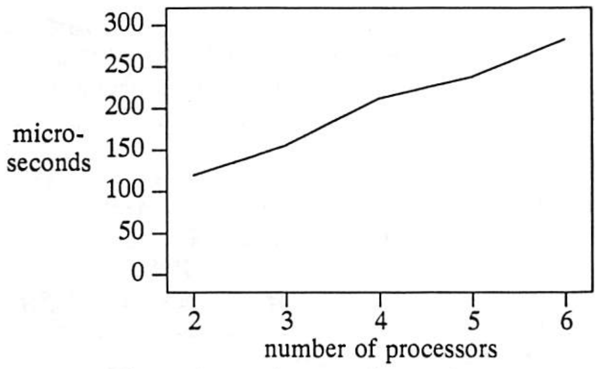


Figure 1: Basic cost of shootdowns

The graph shows that the measurements are not exactly linear. Two possible reasons for this is variances in the interrupt fanout facility on the Sequents and that even a slight disturbance in order of tens of microseonds for each collection of samples will perturb the linearity.

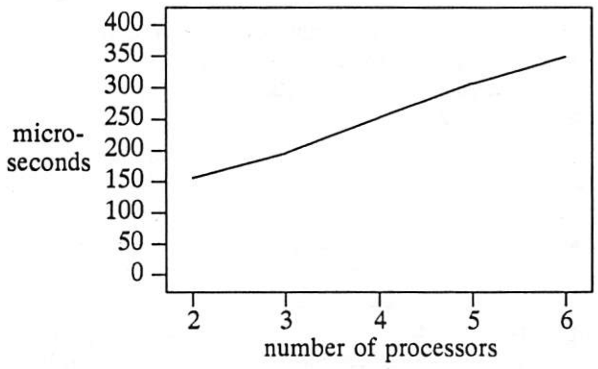


Figure 2: Cost of the Lazy shootdown algorithm (used for kernel address space)

The cost of the lazy shootdown algorithm is illustrated in Figure 2. Note that there is a fixed cost overhead over the basic shootdówn cost in the range of 70 microseconds. The cost of the algorithm per additional processor is about 40 microseconds.

Measurements of the immediate shootdown algorithm were analyzed next. It was measured by running a kernel level test that handcrafted a user address space and spawned as many threads as the number of onlined processors. After the spawned threads waited spinning on a barrier, the parent unmapped a previously mapped page. This would generate a shootdown on all the other processors that were spinning on the barrier. Thus the initiator touches only one page table entry.

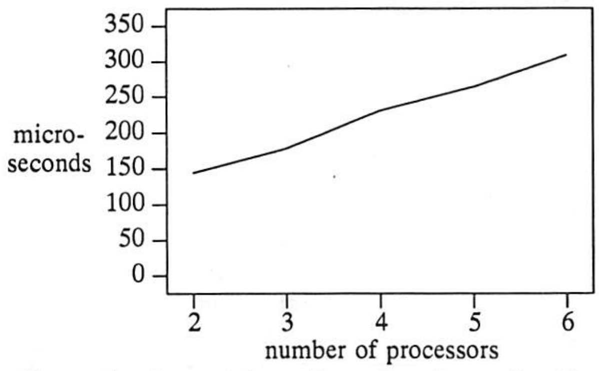


Figure 3: Cost of Immediate shootdown algorithm (used for user address space)

The overhead of the algorithm over the basic shootdown (Fig. 3) cost is about 30 microseconds. The fixed cost of each additional processor for the immediate shootdown algorithm is about 45 microseconds.

The data collected so far indicates that we should be able to scale well in the range of tens of processors for both the lazy shootdowns and immediate shootdowns. The cost of the lazy shootdown algorithm is slightly (about 40 microseconds) more than the immediate shootdown. This is due to all the accounting that is done to update the *cookie* for each

TLB in the lazy shootdown case. Other similar measurements [3] show that the bus contention may become a problem for algorithms that use cross processor interrupts when it deals with processors in the range of 15 - 20.

Another encouraging measurement about the effectiveness of the lazy unload policy of the *segkmem* driver (discussed above) shows that it makes only 1.1 calls to *hat_shootdown* per 100 memory allocation requests. Actual shootdowns will be even less frequent (as can be observed from the explanation of the algorithm). This data was collected by running a kernel level test that allocates and frees kernel memory repeatedly in different sizes. There was no other activity (such as the pageout daemon) in the system when this test was run.

Concurrent fork()/exec()/exit() measurements

Scalability of concurrent inter-address space *fork()/exec()/exit()* operations were measured through a benchmark program[10]. The benchmark consists of the following tests:

- *fork()/exit()* operations with *bss* size ranging from 0 to 192K.
- *fork()/exec()/exit()* operations with *bss* size ranging from 0 to 48K.
- *fork()/sbrk()/exit()* operations with *sbrk* size ranging from 0 to 192K.

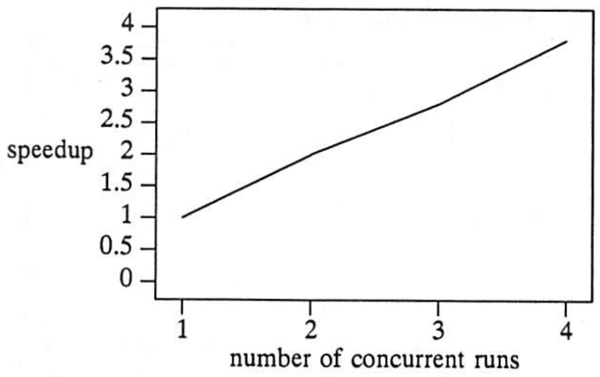


Figure 4: Scalability of fork()/exec()/exit() operations

The measurement process was as follows:

1. Scalability was measured by having a fixed processor configuration (4 processors) and varying the workload of the tests in the benchmark. The workload was varied by executing the same benchmark concurrently from 1 up to 4 times. The speedup was measured by the elapsed time of each work load and measuring it against the unit workload (one run of the benchmark).
2. Each of the above test was repeated for 10 times in a run.
3. The measurement was done on a 4 processor (Intel386 @ 20MHz) Sequent Symmetry machine.

As mentioned earlier, the measurements data (Figure 4) is used only to illustrate the scalability of SVR4.2 and should not be taken as the final performance data of the system.

Conclusions

A model of the HAT layer that is scalable with respect to processors and memory has been described in this paper. This model makes the porting effort simpler without losing sight of the scalability issues. A well defined multiprocessor management interface between the machine independent and the MMU dependent part of Virtual Memory subsystem and simple locking guidelines provide the keys in making a memory management feature portable. The TLB shootdown policy and algorithms in SVR4.2 adapt well to different architectures. With multiprocessor platforms becoming more common, preserving the ease of porting a kernel to different architectures without losing sight of scalability issues will be extremely critical.

Acknowledgements

The design and implementation of the VM subsystem was a joint effort by the SVR4.2 VM team members. The past and present members of this team include: Steve Baumel (also provided *segkmem* measurements), K. Doshi, Mike Lazar of Pyramid Technologies, Joe Lee, and Dave Lennert of Sequent Computer Systems. Dick Menninger was the initial implementor of the SVR4.2 HAT layer. Thanks also to Mike Miracle for his support in writing this paper.

References

[1] Richard Rashid et al., *Machine-Independent Virtual memory management for Paged Uniprocessor and Multiprocessor Architectures*, in IEEE Transactions of Computers, Vol.37, No.8, August 1988."

[2] Robert A. Gingell, Joseph P. Moran and William A. Shannon, *Virtual Memory Architecture in SunOS*, in Proc. USENIX Summer '87 Conference, Phoenix, AR, June 1987.

[3] Black, et. al., *Translation Lookaside Buffer Consistency: A software Approach*, December 1988, CMU-CS-88-201.

[4] Patricia J. Teller, *Translation-Lookaside Buffer Consistency*, June 1990, IEEE COMPUTER.

[5] Michael Y. Thompson et al., *Translation Lookaside Buffer Synchronization in a Multiprocessor System*, USENIX Winter Conference 1988.

[6] 80386 Programmer's Reference Manual

[10]The benchmark program called S is one of the benchmarks used at USL.

Author Information

Ramesh Balan is a Member of Technical Staff at UNIX System Laboratories in the Kernel Development group. He received his M.S. in Computer Science in 1989 from the school of Engineering and Applied Sciences in Columbia University. He can be reached via e-mail at ramesh@usl.com.

Kurt Gollhardt is a consultant at UNIX System Laboratories in the Kernel Development group. He received a B.S. in Computer Science and a B.S. in Electrical Engineering at Washington University in St.Louis. He can be reached via e-mail at kdg@usl.com .

Virtual Window Systems: A New Approach to Supporting Concurrent Heterogeneous Windowing Systems

Rita Pascale, Jeremy Epstein – TRW Systems Division

ABSTRACT

A "virtual window system" (VWS) is a simple model of a window system which can be used to host other more sophisticated window systems. The VWS allows the window systems to share the physical display in a controlled fashion. A VWS is analogous to the virtual machine monitor (VMM) [Madnick74] concept in operating systems, where a single physical computer can run multiple operating systems, each in its own protection domain. Unlike the VMM concept, the window systems supported by the VWS need close cooperation to perform tasks, such as cut and paste between windows of different window systems.

This paper describes the VWS concept, discusses an architecture for a VWS, describes limitations of the VWS concept, discusses some lessons learned from the design and implementation of our prototype, and describes the use of VWSs for various application domains.

Introduction

Users want to run more than one window system (WS) simultaneously on a single platform. The users are a very diverse set whose needs range from debugging to teaching to running a variety of applications. The system developer needs an effective debugging tool for window systems and their applications. The instructor needs a flexible system to teach in as many environments as possible on one machine. The every day user needs to run applications built for more than one window system, a technique which we call "mixed applications".

We propose a general solution to allow many windowing environments to run cooperatively; we refer to it as the "virtual window system" (VWS) concept[1]. A VWS is a simple model of a window system which can be used to host other more sophisticated window systems. The more complicated WSs are considered "guests" on the VWS platform and may be referred to as guest WS. The VWS allows disparate WSs to share the physical display in a controlled fashion and provides a mechanism for communication across WSs.

When compared to similar systems, the VWS is much more versatile. Hybrid WS environments combine two particular systems and provide the ability to run applications from the two specified WSs only. Examples of these hybrid systems are Xvision from VisionWare which combines Microsoft Windows and X; MacX from Macintosh which combines MacOS and X; X-under-NeXTstep from Pencom which combines NeXT and X; and Domain from Apollo which combines Apollo's native WS and X. There are some systems which combine three windowing environments together (i.e., X11/NeWS and SunView), but three seems to be the maximum. By being limited to two or three environments, many of the advantages of the VWS are not possible. The one function these systems fulfill is the ability to run mixed applications and this has been argued as not being a great advantage.

Goals

To meet as many of the various needs of the diverse user group, we need a small, but flexible window system base. To avoid the temptation of building an entire new window system, the size is to be kept small and minimal, using as few primitives as possible. Despite this minimality, we wish to remain flexible. The set of primitives must provide enough functionality to accommodate any window system.

The primitives between the guest WSs and the VWS system can be grouped into connection requests, startup information, input data, and output data. The majority of the primitives deal with changing the display, such as requests to map and unmap windows, update windows, and change the window stacking order. Overall, there are fewer than 20 primitives which is significantly less than the 120 protocol requests in the X server [Scheifler90].

[1]This work is sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA 972-89-C0029.

Details of the Problem

There are a number of problems involved in hosting many environments and protocols on a single platform. The main areas to address are random device accesses and overcoming different protocols.

Access to the keyboard, mouse, console and framebuffer must be regulated. Allowing each WS all input at all times would result in mass confusion since each WS interprets data differently. A mouse click in one environment may pop up a menu, while in another it may cause an application to exit, and in yet another, the data format may not even be valid. Unlimited access to the screen will allow guest WSs[2] to cause chaos by drawing on top of one another's windows, creating a confusing mesh of partial windows.

Another difficulty is cut and paste across WSs. Each system supports a different mechanism through different protocols. A primitive method must be developed that can cut across these various platforms. One drawback of being generic is that unique data formats are not supported. For example, in X [Scheifler90], resource ids (instead of the actual data) can be cut and pasted; this id is only useful within that particular instantiation of the X server and is not meaningful to any other WS process. At the very least, ASCII text can be transferred between all supported WSs, and this is the most common form of data transfer.

[2]By guest window system, we mean a window system environment that is supported and hosted by the virtual window system.

Method

Our solution provides a mechanism to control device access and regulate inter-environment (and intra-environment) communication. The VWS performs these actions through three logical servers: an input manager, an output manager, and a control server. The input manager routes the input to a single designated window system. The output manager displays each window system's output on the screen while handling window overlapping and clipping. The control server is inactive, except for administrative duties. Figure 1 shows the interactions between the VWS and the guest WSs.

Each guest WS must be modified to virtualize its device access. Input is received from the input manager instead of reading the devices directly and drawing is performed in a virtual framebuffer and then sent to the output manager for display. These modifications are necessary since there is no direct access to the hardware devices. The guest WS must also request services of the three VWS servers using minimal primitives. A possible drawback to the virtualized output is that there is no advantage of using intelligent graphics boards unless they can utilize the virtual framebuffers.

In the VWS environment, there is always one active WS which receives all input from the input manager. Any other running system is passive, meaning it can send updates to the screen, but does

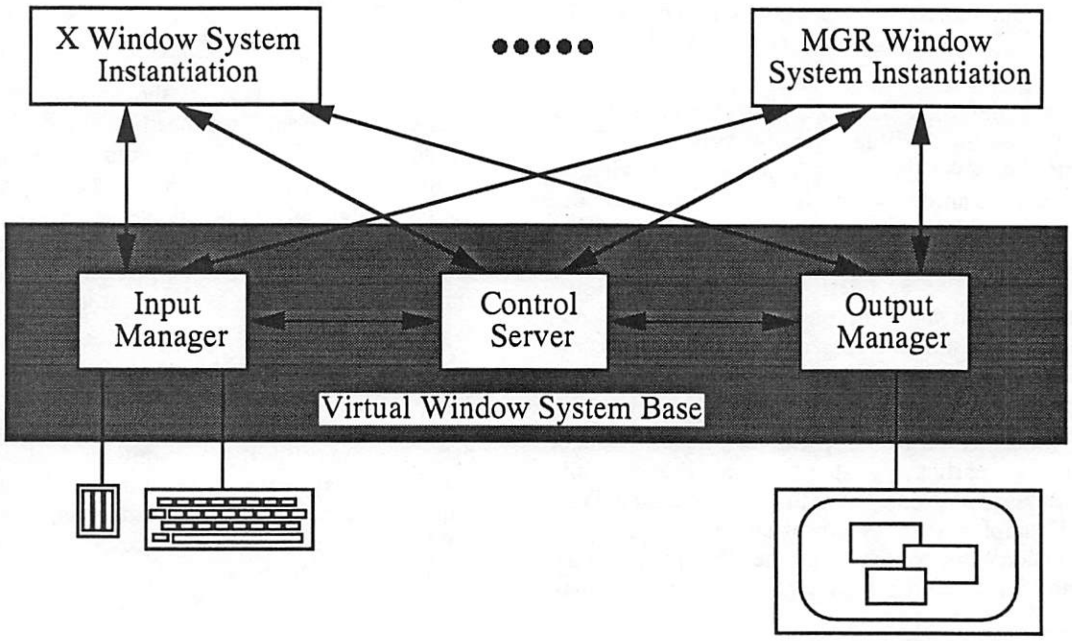

X Window System Instantiation
MGR Window System Instantiation
Input Manager
Control Server
Output Manager
Virtual Window System Base

Figure 1: VWS Interactions

not receive any input. Interpretation of the input is the responsibility of the active window system, meaning it is up to the guest WS to send the input events to the appropriate clients and process them as dictated by its own internal protocol. The input manager's only interaction with the input is scanning for the attention sequence which activates the control server. The control server is activated strictly on certain keyboard input, not clicking on an icon. This is because mouse position (at the time of the click) is up to interpretation per guest WS.

The only primitives from the guest WS to the input manager are connection requests and requests to ring the bell[3]. Input manager primitives to the guest WS provide keyboard and mouse input, initialization data, and notification of selection and deselection by the user.

The output server controls the mapping and unmapping of windows from the various WSs as well as handling stacking order, screen updates and cursor imaging. Because no sophisticated graphics operations are provided, the output manager is much simpler than the output component of an ordinary window system. Performance is the cost of this simplicity, but having such base primitives makes the output manager more adoptive of other WSs. A disadvantage of this scheme is that the screen background is not for use by any guest window system. A mechanism is provided to draw helping lines for placing, moving and resizing windows, but other applications that draw directly on the screen background, such as vine and xroach in the X Window System are simply not supported. These applications are generally decorative and were considered expendable.

The following primitives are supported from the guest WS to the output server:

- connection request
- window map and unmap requests
- cursor position change
- cursor image change
- window update
- raise and lower window requests
- draw dashed box request (for placing, moving and resizing windows)
- load a colormap

The output server primitives to the guests WSs provide initialization data and window map acknowledgements; other requests do not require replies.

The control server starts new WSs, switches between WSs, and provides cut and paste operations between WSs. Cut and paste is the only interaction between windowing environments. The control server provides no interpretation of the data to be pasted. At a minimum, guest WSs must support a canonical text format for interoperability. Additional formats may be supported, but their interoperability is less likely. This may be a disadvantage to systems that have unique data forms, such as resource ids in X.

The primitives sent from the guest WS to the control server are connection requests, cut data and paste requests. The control server responds to the paste requests with the most recent cut data that meets the criteria specified by the guest WS.

With minimal changes, existing WSs can be modified to work within the VWS environment. How closely the window system's implementation is tied to the hardware dictates the amount of change. X is very modular and encapsulates its device usage; therefore, it was quite simple to change. We modified the MIT X11R4 server for Sun hardware to accept input from the input server and display output through the output manager with a few hundred lines of code. Modifying the Macintosh WS would be much more complicated because of its close relationship with its hardware, but we believe that even this can be overcome.

Details on VWS Uses

The VWS exhibits great versatility in its ability to handle a wide variety of needs in a minimal amount of code. VWS neither enhances nor detracts from the given graphical user interface; if the guest WS is poor, it will remain that way. Below we further explain the uses of the VWS.

Debugging Tool

The VWS can be applied to paradigms such as debugging WSs and applications. New versions of the same window system can be tested under the VWS environment. For example, X11R4 can be run at the same time as X11R5 and differences in capabilities of both can be monitored simultaneously. In the same light, the same WS can be run while testing different versions of client applications. For example, one could start three X11R4 servers, and test a different window manager on each. We are able to run the OSF/Motif mwm, OPEN LOOK olwm, and MIT twm window managers simultaneously; each is connected to a different instance of the X server.

Another advantage of the VWS testing environment is that resource grabs are limited to the environment that they were initiated in. In X, a client can "grab" exclusive access to any and all devices, including the server itself. In the VWS system, these grabs are limited to the instantiation of the server that requested the lock. This allows the developer an easier way out of a potential deadlock situation.

[3]Bell ringing is of course an output function, but on our Sun system the bell is part of the keyboard, so we put this function in the input manager.

The VWS is an effective way to test graphical user interfaces; however, performance benchmarking results have no meaning in this environment since all processes are ultimately sharing the same CPU and job scheduler. The main testing advantage is varying visual results due to different configurations and avoidance of detrimental server hangs.

Teaching Tool

Because of its ability to run various environments and configurations, the VWS can be used as a teaching tool for all of those environments rather than requiring one machine for each platform. For comparisons, the VWS can display X11R4 and X11R5 servers and their various applications. Differences are manifested in a more memorable way when they are captured on a single display. The same goes for varying window managers.

Currently, only three window systems are running on our prototype VWS. They are X11R4, X11R5 and Bellcore's MGR[4] With the addition of Macintosh windows, Microsoft Windows and Presentation Manager, this system would be a very useful and inexpensive teaching system. Rather than buying three or four machines, one platform would suffice for all requirements.

Mixed Applications

In today's computing environment there are a half-dozen competing ''standard'' window systems; X, Macintosh, Microsoft Windows, Presentation Manager, and SunView chief among them. VWS allows users to run applications built for more than one window system simultaneously. We can run editres on X11R5, Motif on X11R4, and spot (a pointer tracking program) on MGR. These programs do not run as well (if at all) on the other systems mentioned. X11R4 did not have editres at the time of its release; X11R5 does not support Motif unless it was compiled with the backward compatibility flag; spot is an MGR specific application which does not exist for X11R4 or X11R5.

Again, once other WSs are integrated, the system will become much more useful. For example, with Macintosh and SunView integrated, a software developer could use a document processing program developed for a Macintosh while using development tools designed for the X Window System and SunView system.

Secure Window Systems

Our specific implementation of the VWS is for a highly-secure multi-level window system [Epstein91]. It was developed on a Sun 3/60 running TMach, a prototype trusted operating system based on Mach 2.5. A secure environment is achieved by running multiple instantiations of the X WS and the MGR WS; one at each security level to be displayed on the screen (e.g., one server controls all secret windows on the screen, a different server handles top secret windows). The VWS keeps the workspaces separate and allows cut and paste according to the security policy implemented. This architecture gives us a high degree of trust without relying on the correct functioning of a large and complex window system such as X.

Another advantage of this highly flexible secure VWS base is that the guest WS is entirely untrusted. This means that any window system that can be run on the VWS can be used in a secure environment without having to go through the extensive process of accreditation.

Multi-Processor Environments

Because the VWS consists of several cooperating processes (input manager, output manager, control server, and guest WSs), it is able to use multiple processors without additional investment. For example, the X server could run on one processor while the MGR server runs on a different processor.

Results

The design and implementation of the VWS took two man years. Modification of the X11R4 server required less than four months for a junior programmer to incorporate the necessary changes. Upon the release of X11R5, modifications were adapted in less than a week. We adopted Bellcore's MGR window system to the VWS with less than a month. Using ports and multi-threading, fundamental aspects of Mach-like operating systems, attributed to the bulk of the modified code. Much of the low level communication code in the X WS had to be rewritten to use ports; using an operating system that supported sockets would have saved implementation time.

Runtime improvements can be made by using faster hardware and implementing the VWS in an environment that supported sockets (rather than ports as in TMach) and shared memory. Other benefits to using sockets over ports is that sockets can be prioritized. Without this ability, previously simple processes had to become multi-threaded to force prioritization on port message retrieval. For example, our X server has one thread per client in addition to several control threads. While multi-threading has its benefits in allowing many activities to occur at once, there is a performance cost in context switching, and a general overhead burden. Despite our poor hardware configuration, the system is not intolerable. With some enhancements, including those mentioned above, performance would be greatly improved.

[4]MGR [Uhler88] is a freely available window system. It is far less flexible than X, but it is faster and smaller.

Running benchmarks on the VWS for comparisons with unchanged WSs has proven to be much more difficult than expected. Currently we have some preliminary results from x11perf that show the X11R4 server running under VWS yields 50 to 75 percent of the runtime speeds compared to the X11R4 server running directly on the hardware. Window manipulations (such as raises, lowers, circulates, maps, and unmaps) performed comparably whereas graphics operations did not perform as well. In the graphics area, the VWS system compared best on tests of direct copies rather than stippled patterns[5]. Also, simpler requests like lines and rectangles performed significantly better than more complex shapes like circles and ellipses.

Our VWS implementation is less than 20,000 lines of heavily commented C code (about 6,000 statements). A significant fraction of that is due to security requirements. By contrast, an X11R4 server and Motif window manager total about 400,000 lines of code, including support libraries.

Our implementation can be improved with faster machines, hardware that can handle many framebuffers, tuned operating systems as well as a number of other things. Despite the added overhead, the performance is acceptable for the typical user. The more intensive the load, the more the performance will downgrade.

Conclusion

The architecture and implementation of the VWS system has achieved our goals of minimality and flexibility. As a proof of concept experiment, the notion of a VWS has proven itself useful for building trusted window systems. We feel it is applicable in other problem domains as well, and offers significant advantages over alternate architectures. There are limitations to our implementation, but may be acceptable, given the payoffs of being able to operate in a heterogeneous environment. Also, some of our limitations are specific to security. If shared data between WSs is allowed, as is the case in unsecure systems, the memory usage and performance would improve immensely.

Future work includes research into different hardware, different operating systems and more guest window systems.

References

[Madnick74] *Operating Systems*, Stuart Madnick and John Donovan, McGraw Hill, 1974.

[Scheifler90] *X Window System*, Second Edition, Robert Scheifler and James Gettys, Digital Press, 1990.

[Uhler88] Stephen Uhler, *MGR – C Language Application Interface*, Bell Communications Research, 1988.

[Epstein91] Jeremy Epstein, et. al., ''A Prototype B3 Trusted X Window System'', published in *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio TX, December 1991.

[5]Stippled patterns are as stencils to indicate where to draw and where not to draw.

Availability

A series of technical papers are available from the authors on our VMS implementation to support a highly trusted version of X. The software itself is not available at this time.

Author Information

Rita Pascale is a Programmer on TRW's Advanced Computing Systems project building a highly trusted version of the X Window System. She holds a B.S. in Computer Science from Virginia Tech. Her U.S. Mail address is 1 Federal Systems Park Drive, Fairfax VA 22033. She can be reached electronically at pascale@trwacs.fp.trw.com.

Jeremy Epstein is the Lead Engineer on TRW's Advanced Computing Systems project building. In his previous life, he was a lead engineer with Addamax developing trusted UNIX systems. Jeremy has been working with UNIX since Version 6, and still refuses to use ''csh''. He holds a B.S. in Computer Science from New Mexico Tech, M.S. in Computer Sciences from Purdue University, and is working on a Ph.D. in Information Technology at George Mason University. His U.S. Mail address is 1 Federal Systems Park Drive, Fairfax VA 22033. He can be reached electronically at epstein@trwacs.fp.trw.com.

A Discipline of Error Handling

Doug Moen

ABSTRACT

In the UNIX world, exception handling mechanisms for error handling are often discussed, but seldom applied. This paper describes a disciplined approach to error handling that was refined over a 3-year period during the development of a medium-large (200K line) toolkit written in C under UNIX. We describe both a portable exception handling system, written in C, and a methodology for using it which encompasses coding style, documentation, and testing issues.

Introduction

The C language, as defined by Kernighan and Richey and by the ANSI C standard, is rather weak on error handling. The only standard error handling facilities provided are the global variable errno and the convention that certain functions (such as malloc, etc.) return a distinguished value when they fail, possibly setting errno.

This is not a very good basis for error handling.

It isn't good for application programmers, because explicitly checking the return values of functions that might fail, and propagating the error, is a lot of work, and clutters up code. Even conscientious programmers have been known to write programs that fail to check the return value of every call to printf. As a result, there are a non-trivial number of C programs in existence which fail to properly report error conditions [Darwin 85].

The standard approach to error handling isn't good for library implementors, either. One problem is that the existing set of error numbers is not extensible; thus, you can't use the standard functions perror(), strerror(), etc., with locally defined error codes. Another problem is that a single integer (errno) does not really contain enough information to completely describe an error. Usually there is additional contextual information (such as the name of the file that couldn't be opened, the number of bytes that were successfully written before an error occurred, the line number on which the error was detected, etc.) that needs to be associated with an error.

We faced these problems when we set out to build the EMS image processing toolkit in 1989. We were building a large library of functions for building image processing applications, and we wanted our library to support the construction of robust applications. We wanted our library to support good error handling. So we defined a comprehensive approach to good error handling, and wrote a small library of functions to support our error handling discipline.

There is more to good error handling than simply language or library support for raising and catching exceptions. In this paper, I will explain what errors are, describe the design issues in representing and reporting errors, and discuss how our error handling approach affects coding style, documentation and testing.

Two Kinds Of Errors

EMS distinguishes two kinds of errors that can be detected by library functions: faults and failures.

A fault condition is the failure of an assertion or sanity check. By definition, a fault always indicates the presence of a bug in a program. Faults checking is not part of the contract between the function and its caller, and faults are reported by aborting the program. Some kinds of fault checking (e.g., comprehensive data structure integrity checks) are expensive to perform, but are useful during debugging. Because client code is not allowed to depend on the existence of fault checks, expensive fault checks can be conditionally compiled based on a DEBUG option, without affecting the correctness of any program.

A failure is an abnormal but anticipated condition such as resource exhaustion, permission denied, or a syntax error, which prevents the function from carrying out its job. Failures differ from faults in that failure reports are part of the contract between the function and its caller. Failures are reported by reporting an error back to the caller. An out of memory condition that causes malloc() to return NULL is a simple example of a failure.

Sometimes it is difficult to decide if a particular condition (e.g., an illegal argument value) should be classified as a fault or a failure. My rule of thumb is that it should be possible, using the library, to write programs that never generate fault conditions under any circumstances. Suppose that the exceptional condition is the detection of a syntax error in an input file, or in a character string that might have originated from outside of the program. In this case, the condition should be classified as a failure, rather than as a fault. If it were classified as a fault, then a program written to avoid generating faults is obliged to scan the input beforehand to ensure the absence of syntax errors.

Reporting Faults

In EMS, faults are reported by calling the function fatal() with a printf-style argument list. fatal() aborts the program by calling a handler function registered using at_fatal(). If no handler function is registered, or if the registered handler function returns to its caller, then fatal() prints a message to stderr and calls abort() to obtain a core dump.

The decision to report faults by terminating the program was a controversial one. It was made for the following reasons:

- During a development cycle, the best way for a program to report a fault is to immediately dump core. The core file can then be used for post-mortem debugging.
- If a function signals a fault by reporting an error to its caller, then this error report becomes a de facto part of the function's contract with its caller, and it becomes possible to write code that depends on intercepting fault reports. We don't want fault reports to be part of a function's interface, because checking for faults can be expensive, and we would like to have the option of turn off fault checking using compile-time options, in order to speed up the code. Turning off fault checking should not change the semantics of a correct program.
- Assertions and sanity checks are easier to code, and are more likely to be used, if developers don't have to worry about cleaning up after a failed assertion.

An unfortunate consequence of aborting a program when a fault is detected is that we don't support fault tolerant programs of the sort that try to keep running after a fault has been detected (e.g., see [Meyer 88] and [Randell 75]). This isn't quite as bad as it sounds, because EMS has two provisions for supporting fault tolerance. First, programs can register a handler to save their state when fatal() is called. Second, EMS has facilities for automatically restarting crashed servers. Thus, an application that consists of a network of processes can be made to keep running even when individual components crash.

Reporting Failures

The EMS failure reporting system is based on the following principles:

1. Error propagation. In a typical application, errors are detected at a low level (for example, in a library function), and handled at a higher level (for example, in an application program which calls the library). Only the higher level code knows how to handle the error; this might be to print an error message on the terminal, display an error window, or terminate the program. It is inappropriate for low level code to take these actions. Therefore, when an error is detected, a description of the error is propagated from the low level code, where it is detected, to the high level code (higher in the call stack) where it is handled.
2. Error values. In the traditional C approach to error handling, errors are described by a single small integer. This is inadequate; if the high level error handler needs to display an error message to the user, then it usually needs more than just the type of error (e.g., ''syntax error''): it usually needs some information about the context in which the error occurred (e.g., file name + line number). In our system, errors are described by an Error structure which contains both an error id (which is an integer) and character string data which can be used to print an informative message.
3. Exception handling. In the traditional C approach to error handling, a function returns a distinguished value (such as NULL or -1) when it gets an error, and sets the global variable errno with an error id. It is the responsibility of the caller to check for an error return status, and either handle the error, or propagate it up the call stack. Although this approach is simple, it is also error prone: it is very easy for a programmer to be lazy, and omit checks for error return codes. If these checks are omitted, the program may go wrong in a catastrophic way when an error occurs. Our solution to this problem is to raise an exception when an error occurs. When a function raises an exception, it immediately terminates, its caller terminates, and so forth, until an exception handler is found. If no exception handler can be found anywhere on the call stack, the program terminates with an error message. Under this scheme, a programmer must take positive action to prevent his function from being terminated by an error. The worst that can happen if he forgets to check for errors is that, at the library level, temporarily allocated resources may not be freed, and at the application level, the program may terminate with a message.

Error IDs

The most important component of an error description is an error 'id', which identifies the type of error that occurred. Error ids are used by error handlers to distinguish different types of errors. If the error id has a short string representation, then it can also be printed out as part of the error message, and used as a key by the end-user to look up a verbose description of the error in external documentation.

Any system for defining error ids needs to deal with two issues. First, it should be possible for programmers to define new error ids without editing a master table somewhere, and without conflicting with error ids defined by other libraries. Second, it should be possible to define sets of related error ids so that error handlers can check error ids for membership in a group, as an alternative to enumerating a long list of error ids that are to be handled in the same way.

The UNIX/ANSI C system of error ids (as defined by <errno.h> and sys_errlist) is not extensible, and provides no support for grouping.

Programming languages with built-in exception handling systems provide the ability to define new error ids as a matter of course, but not every such language provides a way to define groups of errors. In both ANSI C++ and in Common Lisp, it is possible to organize error types in trees or DAGs using single and multiple inheritance [Ellis 90, Steele 90].

In the exception handling system devised by Allman and Been [Allman 85], error ids are character strings, and error handlers can use glob-style pattern matching on error ids.

In EMS, we chose a system similar to that used by ANSI C and UNIX. Error ids are represented by integers, which means that error handlers can use switch statements to distinguish between different errors. When an error description is printed, the error id is used as an index into a table of message strings.

In order to support extensibility, we support multiple error tables. Each error id is a 32 bit integer with a 19 bit table id, and a 13 bit offset, which is an index into the table. The table id is computed from a table name: this is a string of between one and four lowercase letters which is converted to an integer using a variant of base 26 encoding.

Each error table is maintained as a text file that defines 4 records for each error:

- An error name, which is a C identifier like ENOENT or E_SYNTAX. The error name is used to #define a constant. In addition, it is printed by err_print_x for the benefit of both users and programmers. Users can use the error name to look up additional information about the error in externally provided documentation.
- A severity level, which is one of the constants ERR_FAULT, ERR_FAIL or ERR_RETRY. ERR_FAULT denotes a fault in an external program or subsystem, ERR_FAIL denotes a permanent failure, and ERR_RETRY indicates a temporary condition that will clear up by itself with no intervention.
- A short, one line message, analogous to the messages in sys_errlist. This message is printed by err_print_x().
- A verbose description of the error, typically one paragraph in length. This is incorporated into external documentation for errors.

This text file is processed to generate a .h file, a .c file, and a documentation file. The .h file contains one #defined constant for each error id, plus an external declaration for a global variable of type ErrTab_t[], representing the error table. The .c file contains the definition of the error table, which contains, for each error id, the message string, the severity level, and the error name, represented as a string.

By convention, in an error table named "foo", each error id has the prefix "EFOO_", the error table variable is called "foo_errs", and the header file is called "foo_errs.h".

System error codes, as obtained from the header file <errno.h> or the global variable errno, can be converted into valid error ids by casting them to type long. The corresponding error table is called sys_errs. All system errors are arbitrarily assigned the severity level ERR_FAIL.

The only provision that EMS has for defining groups of error ids are the fixed set of error severity levels. Although this has proven adequate for our needs, it is not very flexible.

Error Values

When a function fails, it is responsible for conveying a description of the error to its caller. This description can be used in several different ways: it might be interpreted by an error handler which needs to take different actions on different errors, it might be used to construct an error message which will be displayed to a human user, and it might be transmitted to another process on the network (as when a server notifies a client of an error).

Since the error description might be interpreted by an error handler, it needs to contain an error id, plus any additional contextual information that might be needed by the error handler, such as the number of bytes that were successfully written before the error, the line number on which the error occurred, etc. Since the error description might be used to construct an error message, it needs to contain all of the information necessary to make this possible. Finally, since the error description might be transmitted to another host on the network, it needs to be represented in such a way that it can be transformed into a machine-independent byte stream for transmission purposes, then reconstituted by the recipient.

Let's consider the case of constructing an error message. Error messages are seen by two classes of people: end users, who don't understand the internals of the program that failed, and gurus/implementors who do.

For end users, the message should be phrased in high level terms, and provide enough information so that it is possible for the user to determine a corrective course of action. In practice, this means a one-line message containing a phrase describing the problem, the context in which the error occurred (e.g., file name, line number, etc.), and an error name which can be used as a key to look up a more complete description of the error in external documentation.

For gurus and implementors, the message should contain additional information about what went wrong at the implementation level of the program, because this information might be needed for debugging, or in order to fix a problem somewhere in the system. This lower level information can be generated naturally, as a consequence of the way that error descriptions are generated. When an error is detected, a description of the error is passed down the call stack through one or more levels of function calls until it is finally handled. Now, consider that each function is obligated to present an error interface which makes sense in terms of the abstraction that it implements. When an error description is forwarded through an abstraction boundary, it is sometimes necessary to reinterpret the error in terms of the current level of abstraction, which means supplying a new error id, and new contextual information. The lower level error description which is being supplanted need not be thrown away; this low level description of the error might be of use to gurus and implementors when it is presented in an error message.

Putting all of these requirements together, we find that an error description should consist of a stack of one or more error interpretations. The interpretation on the top of the stack is a high-level description of the error, while the interpretation at the bottom of the stack describes the error in terms of the function in which the error was first detected. Each error interpretation contains an error id, plus additional contextual information. We need operations on error descriptions for printing or displaying them as human readable messages, and for transmitting them across the network to other hosts (which may have different byte ordering, different floating point representations, etc.).

In EMS, error descriptions are represented by values of type Error_t. An Error_t contains a stack of error interpretations. Each interpretation contains an error id, the error name represented as a string, the short message associated with the error id, the error severity level, and a character string containing contextual information which is generated at run time, when the error is detected.

Note that the contextual information associated with each interpretation is represented as a single character string. This is a good representation for printing error messages and for transmitting error descriptions across the network, but it is not a convenient representation for error handlers which wish to access and interpret the contextual information. In fact, the current implementation of EMS provides no programmatic way to access the contextual information at all! Although this limitation has not caused any problems so far, I think it should be fixed.

The stack within an Error_t has a fixed maximum size of 6 entries, and there is a fixed amount of space for storing character string information. If you try to push more than 6 interpretations onto the stack, then the bottom interpretations are thrown away. If you try to store an oversize context string in an interpretation, then it will be truncated. We chose to use fixed sized arrays for representing Error_t's because we wanted to avoid the use of dynamically allocated storage. We did not want to run out of heap space while attempting to report errors in a low memory situation, and we did not want the additional complication of having to explicitly free the storage associated with Error_t's within an error handler. Instead, we use automatic storage for all Error_t's. In practice, the stack size limitation is not a problem, and neither is the occasional truncation of context strings.

Operations On Error Values

When an error is first detected, an error value is initialized with a single error interpretation, consisting of an error id, the associated error name, severity level and short message, and an optional context string. If the error were printed at this point, it would look like the line in Figure 1, below.

An error value can be initialized to contain a single interpretation by calling err_set() (see Figure 2, below).

```
context string: short message (error name)
```

Figure 1: Error message format

```
void err_set( Error_t *err, ErrTab_t *tbl, long id, char *fmt, ... )
```

Figure 2: Prototypical `err_set` call

The fmt argument is the beginning of a printf argument list, which is used to construct the context string. See Figure 3 for an example.

When an error value passes through an abstraction boundary, it may be appropriate to push a new interpretation onto the stack which describes the error in higher level terms. This is done by calling err_push(). Alternatively, instead of pushing a new interpretation onto the stack, you can choose to push an annotation by calling err_note(). An annotation is a character string which is added to the top level error interpretation without changing the error id. Annotations are used to add information to to the error message seen by a user, without changing the error description from the point of view of an error handler. For example, the sequence of calls in Figure 4 would create an error description which would be printed as shown in Figure 5.

For completeness, we also supply err_clear(), which initializes an error description to an empty stack, and err_pop(), which pops the top level interpretation, so that the error id underneath can be accessed.

An error handler which needs to distinguish between different types of errors can query an error using the functions err_id() and err_level(). These functions return the 32 bit error id and the error severity level, respectively, of the top level interpretation within an error. The severity level is one of the following three constants:

```
#define ERR_RETRY 1
#define ERR_FAIL  2
#define ERR_FAULT 3
```

These constants are ordered by severity so that < and > tests can be used on them.

An error can be printed by calling err_print_x(). This function prefixes each line of output by the program name and a colon. If it is desired to display an error message within a dialog box, then you can call err_string_x() to obtain a character string representation of the error message, with embedded newlines, but without the program name prefixes.

Errors can be written to and read from a network connection in machine independent binary format by calling err_write_x() and err_read_x(). An unresolved problem with these functions is that UNIX error numbers change their interpretation from one machine to another.

Internationalization

In multilingual environments, it is necessary to ensure that error messages are displayed in the correct natural language. Of course, error message strings are not the only strings that need to be translated, and it is appropriate to regard error handling and internationalization as orthogonal problems that deserve separate and independent solutions.

Different operating environments provide different solutions to the problem of internationalization. In the Macintosh and Windows environments, character strings that need to be translated are stored in resource files which are shipped with the application; the buyer must purchase a copy of the application which has been translated to the desired language. In the OSF environment, the NLS package supports a run-time choice of several different natural languages by setting the LANG environment variable; once again, strings containing natural language are stored separate from the programs themselves.

In the EMS toolkit, we have a high-level abstraction for string translation which can map cleanly onto the facilities provided by all of the above environments. The construct

```
XSD(string-literal)
```

is replaced by string-literal on systems that don't support internationalization, and is replaced by a function call which returns a pointer to the translated

```
Error_t err;
err_set(&err, sys_errs, (long)ENOMEM, "Couldn't allocate %d bytes", size);
```

Figure 3: Creating an error value

```
Error_t err;
err_set(&err, util_errs, E_EOF, "File descriptor %d", fd);
err_push(&err, util_errs, E_CORRUPT, "");
err_note(&err, "Error while reading file \"%s\"", filename);
```

Figure 4: Creating another error value

```
Error while reading file "foo"
> Corrupted data file (E_CORRUPT)
> File descriptor 3: Unexpected end of file (E_EOF)
```

Figure 5: A printed error value

string on systems that do. In other words,

```
XSD("Hello")
```

returns a pointer to the string "Bonjour" if compiled in an environment that supports internationalization, and the current language is French. The construct

```
XSI(string-literal)
```

returns a static initializer for a structure of type XS_t. The function xs_pgets() takes a pointer to an XS_t, and returns a pointer to the translated string. The XS package is implemented using a modified C preprocessor which maps string literals onto the appropriate magic incantations for fetching the translated version of that string from the appropriate resource file or database.

Thus, internationalization of error messages within the EMS environment becomes a trivial problem. The short error messages within each statically initialized ErrTab_t variable have type XS_t, and are initialized using the XSI macro; this is transparent to the programmer. Programmers must be careful to use the XSD macro to translate printf() style format strings that contain natural language. That's it.

Throwing and Catching Errors

In the early days of EMS, functions reported failures by returning a special error value. This approach was abandoned because of the code clutter caused by checking nearly every function call. It was replaced by a portable exception handling mechanism using the termination model.

A function signals failure by building an error value, then throwing it to its caller as shown in Figure 6. As a shorthand, we supply the function throw_err_x() which allows the above code to be written in a single line (see Figure 7). The effect of throw_x() or throw_err_x() is to raise an exception: this causes a non-local jump down through the call stack to the nearest exception handler. If no exception handlers are active, then the error is printed to stderr, and the program exits with status 255. As a result of this default behaviour, simple utility programs that exit with an error message when any error occurs are very easy to write.

Exceptions may be caught using the control structure shown in Figure 8. The "THEN_TRY ..." clause may be repeated zero or more times. The identifier argument to CATCH is used to declare a variable of type Error_t* which is local to the exception handler. A TRY .. END_TRY block is syntactically a compound statement, and may be written anywhere a statement is legal in C. They may even be nested.

A TRY .. END_TRY block is evaluated in the following manner. First, each body is executed in sequence. If an exception is raised during the execution of a body, then control is immediately transferred to the beginning of the next body. After all of the bodies have been executed, one of two things happens. If an exception occurred during the execution of any body, then the handler is executed with the Error_t* variable pointing to the first error that occurred. If none of the bodies raised an exception, then the handler is skipped.

For example, the following code:

```
TRY
        foo_x()
CATCH (err)
        err_print_x(err, stderr);
END_TRY
```

calls foo_x(); if it raises an exception, then the exception is caught, and the error is printed to stderr.

If you use break, continue, goto or return to break out of the body of a TRY or THEN_TRY clause, then you must call TPOP() before

```
Error_t err;
err_set(&err, sys_errs, (long)ENOMEM, "Couldn't allocate %d bytes", size);
throw_x(&err);
```

Figure 6: Creating an error value and throwing it to its caller

```
throw_err_x(sys_errs, (long)ENOMEM, "Couldn't allocate %d bytes", size);
```

Figure 7: Combined set and throw functions

```
TRY
        .. body1: statements that may raise an exception ..
THEN_TRY
        .. body2: more statements that may raise an exception ..
CATCH (identifier)
        .. exception handler ..
END_TRY
```

Figure 8: Exception catching

transferring control. Otherwise, the stack maintained by the exception handling system will be damaged. TPOP() can only be used to break out of a single TRY statement; it cannot be used to break out of several nested TRY statements at once. For example:

```
TRY
    if (bar_x() == 0) {
        TPOP();
        return;
    }
CATCH (err)
    err_print_x(err, stderr);
END_TRY
```

The exception handling system described in this section is written in portable C, with the help of setjmp(), longjmp() and the C preprocessor. Although its genesis is independent, the implementation is similar to that used by Roberts [Roberts 89]. The need for the TPOP() macro is regrettable, and is an occasional source of bugs. A purpose built preprocessor could eliminate the need for TPOP(), and could also generate slightly better quality C code for the TRY statement.

Cleaning Up After Errors

It is the responsibility of every library function to clean up properly after detecting an error. There must be no memory leaks, file descriptor leaks, or data structures left in an invalid state after an early error exit.

Consider this function (note that mem_alloc_x() is a wrapper for malloc() that raises exceptions, and mem_free() is a wrapper for free()):

```
void
foo_x()
{
    Foo_t *f1, *f2;

    f1 = mem_alloc_x(sizeof(Foo_t));
    f2 = mem_alloc_x(sizeof(Foo_t));
    ... body of foo_x ...
    mem_free(f1);
    mem_free(f2);
}
```

If the first call to mem_alloc_x() fails, then foo_x will be immediately terminated by an exception. This is the desired effect, and no error handling code is required within foo_x to make this happen. However, if the second call to mem_alloc_x() fails, then there will be a storage leak, because foo_x will exit without freeing f1. Similarly, if the body of foo_x (as represented by the ellipsis) is capable of failing, then neither f1 and f2 will be freed. Our solution to this problem is to use the coding style in Figure 9.

The TRY clause contains resource allocation and the body of foo_x. The THEN_TRY clause frees resources; it is executed whether or not the body fails. The reason that we initialize f1 and f2 to NULL is so that the calls to mem_free() in the THEN_TRY clause will work even if an exception is raised in one of the calls to mem_alloc_x(). mem_free() is guaranteed to ignore a NULL argument, unlike free() on some UNIX systems.

```
void
foo_x()
{
  Foo_t *f1, *f2;

  f1 = NULL;
  f2 = NULL;
  TRY
    f1 = mem_alloc_x(sizeof(Foo_t));
    f2 = mem_alloc_x(sizeof(Foo_t));
    ... body of foo_x ...
  THEN_TRY
    mem_free(f1);
    mem_free(f2);
  CATCH (err)
    throw_x(err);
  END_TRY
}
```

Figure 9: Cleaning up after an error

This kind of analysis (for resource leaks) has to be performed every time a library function is written. In order to make the analysis and coding easier to perform, we strictly enforce two conventions. First, all library functions that are capable of raising exceptions have names suffixed by _x. Second, all deallocation routines are required to ignore a NULL argument. The _x convention has proven to be quite useful, because it is otherwise very difficult to tell whether or not a particular stretch of code is capable of raising exceptions, and this is critical for resource leak analysis. It is not convenient to assume that every function call could raise an exception, because we make heavy use of access macros to replace direct access to structure members, and these usually don't raise exceptions.

Documenting Exceptions

One of the rules that we have tried to enforce is that the error interface provided by each function must be fully documented. After all, this error interface is part of the contract that the function makes with its callers, just as surely as the result and argument types are.

We have run into several problems trying to achieve this goal.

The first problem arises from the fact that it is difficult to document the error interface of high-level functions if the low-level functions that they call don't have well-defined error interfaces. Unfortunately, we have this problem with the UNIX system calls. On many UNIX systems, the set of errors generated by each system call (and the semantics of system calls when an error is detected) is only partially documented. Furthermore, the error interface for system calls is not portable: it changes from one system to the next. Consequently, EMS library functions that perform I/O currently have a poorly documented, system dependent error interface, and writing error handlers for code that does I/O sometimes involves experimentation to find out the names of the error numbers of interest, combined with #ifdefs to check for different error numbers on different machines. The obvious solution to this problem, which we haven't had time to pursue, is to define a portable error interface for each system call, then to work out the mapping from system error ids to portable error ids for each system call and machine type.

The second problem arises from the fact that the error interface for a function tends to be defined as the union of the error interfaces for all of the functions that it calls. Not only does this lead to large, cluttered error interfaces, but it also means that error interfaces tend to change over time as a result of maintenance, and thus the documentation for error interfaces tends to drift out of sync with reality. This can be viewed as a documentation and maintenance problem, in which case the solution might be to attempt to generate the documentation for error interfaces automatically, using a code analysis tool. However, this problem can also be viewed as a design problem: in some cases, programmers are not making the effort to design a high-level, abstract error interface that matches the abstraction provided by the function; instead, they are letting the error interface default to whatever their code does.

Many strongly typed languages with built-in exception mechanisms either permit or require you to declare the error interface of a function as part of its type. (C++ and Modula-3 are examples.) This declaration consists of a fixed list of error ids. In our experience, life is not that simple. If you are doing object oriented programming (as we do), then what you will find is that different subclasses of a base class A will implement different error interfaces for the virtual function inherited from A. For example, we have a class called Stream (similar to a stdio FILE). The set of errors that can occur in (eg) the write_x virtual function depends on which subclass of Stream you are writing to. As a result, the error interface for a function that takes a Stream as an argument depends on which subclass of Stream is actually passed in. Language designers might wish to consider polymorphic exception sets within function signatures to deal with this issue.

Testing

An important part of our error handling package (and a topic rarely discussed in the literature on exception handling) is testing. Full adherence to the error handling discipline described here adds a noticeable amount of complexity to our code, mostly in the form of exception handlers within library functions which deallocate resources and restore data structure invariants before forwarding an exception. These exception handlers are rarely executed in production code, and are therefore a fertile breeding ground for bugs. Fortunately, we have developed a simple, yet powerful mechanism for testing these exception handlers. This mechanism works by triggering fake exceptions in low-level functions under the control of command line arguments. These fake exceptions set off a cascade of intermediate exception handlers, thereby exercising them. When used in conjunction with a library regression test program, this mechanism can be made to exercise all of the exception handling code in a library.

A control point for triggering a fake exception is placed by calling the macro xtrap_x(), which takes an error id as an argument. These control points are placed in low level library functions, at points where an exception could potentially be raised by natural means. Only a handful of calls to xtrap_x are needed in EMS; the single call to xtrap_x in mem_alloc_x (our wrapper for malloc()) is sufficient to test 75-80% of our exception handling code.

EMS has a mechanism for passing debug options into a program for use by library routines. These debug options can either be placed in an environment variable, or they can be supplied as arguments to the command line flag -V, which all EMS programs support. The xtrap mechanism is triggered using the command line argument -V xt=i, which causes the i-th dynamic invocation of xtrap_x to raise an exception. The command line argument -V xc causes a program to run to completion, then print out the number of times the xtrap_x macro was executed. To exercise a library, we write a test program which exercises all of the functions in the library. Then we run the program with -V xc, which gives us the number n. Finally, we run the program n times (from a shell script), supplying the i-th invocation with the argument -V xt=i. This is usually sufficient to exercise 99% of the exception handling code in the library (assuming that the test program is written to provide full coverage of the library). When this testing technique is combined with the debugging version of our storage allocator (which detects bad calls to free and storage leaks), we can detect most problems involving improper releasing of resources in exception handlers.

Evaluation

The EMS approach to error handling is a success. Our code is robust, well behaved, and testable, and programmer acceptance of the approach is high. There is an initial cost in learning how to use the error handling system, especially for junior programmers, who can be a little uncomfortable with the fact that most functions they call are liable to perform a non-local jump upon encountering an error. However, once programmers get past the initial learning hump, they seem to like the system, since using it leads to cleaner looking code that is less obscured by error handling than equivalent code which checks each function call for error return values.

Under the old regime of signalling errors by return codes, the most common bug is failing to check the return code. This can lead to nasty behaviour, like core dumps. Under the new regime of using exceptions to signal errors, the most common bug is failing to catch exceptions for the purpose of freeing resources, and this leads to resource leaks[1]. The new style of bug has fewer harmful effects on the overall robustness of the code.

There is still room for improvement. A more flexible method for organizing error ids into a hierarchy, so that programmers can test an error for membership in a group, and a way of associating integer and string parameters with an error, would be welcome improvements. So would a lint-like tool for detecting common bugs in exception-handling code, and a tool to automate the documentation of error interfaces by scanning source code.

The EMS error handling system compares quite favourably to other C exception handling packages. Its greatest strength is the Error_t structure, which incorporates the novel idea of a stack of error interpretations, and provides standard ways to print arbitrary error descriptions, and to transmit them across an IPC connection. Roberts' system [Roberts 89] provides a mechanism for raising and handling exceptions very similar to the EMS system, but errors are represented by a pair consisting of an error id and an integer parameter. Allman's system [Allman 85] provides a flexible way to organize error ids into groups, has character string parameters which are accessible to error handlers, and integrates UNIX signals with the exception system. On the minus side, his system requires assembly language support, and has a much less convenient syntax for exception handlers. Allman also supports the more general resumption model of exception handling. My feeling is that terminate and resume style exceptions should be signalled and handled by different mechanisms; see the appendix.

[1]Using the debug version of the EMS memory allocator, resource leaks are not difficult to diagnose, since we provide a facility for listing the memory blocks which are still allocated at program exit time; this list gives the file name and line number of the call to mem_alloc_x for each allocated block.

A Plea For Standardization

The benefits of an exception handling system are strongest when it is used everywhere. Accordingly, the EMS utilities library contains exception-raising replacements or wrappers for many of the most commonly used C library functions. We even went so far as to implement a complete replacement for stdio with better error handling[2] (and of course better performance). Unfortunately, we don't get the full benefit of our error handling system when we use libraries, such as those for the X Window System, that were written by other people. Because the C error system (errno, perror, etc.) is not extensible, every library implementor is forced to create their own error handling system, and application programmers are stuck with the job of knitting these different error handling systems together.

We think the world would be a better place if the C community had a standard, extensible, and well designed system for fault and failure handling which all library implementors used. This probably won't happen in the C community, but there is still hope for new languages such as C++. Implementations of C++ that support exception handling are just now starting to appear. Unfortunately, syntax for raising and handling exceptions is not enough: there should also be standard conventions for using it, a standard way to print errors, and exceptions raised in all standard library routines upon failure.

References

[Allman 85] Eric Allman and David Been. "An Exception Handler for C, " Proceedings of the Summer 1985 USENIX Conference. Portland, Oregon, 1985.

[Darwin 85] I. Darwin and G. Collyer. "Can't happen -or- /* NOTREACHED */ -or- Real Programs Dump Core," Proceedings of the Winter 1985 USENIX Conference. Dallas, Texas, January 1985.

[Ellis 90] Margaret A. Ellis and Bjarne Stroustrup. The Annotated C++ Reference Manual. Addison Wesley, 1990.

[Goodenough 75] John B. Goodenough. "Exception Handling: issues and a proposed notation," Communications of the ACM. Vol. 18, no. 12, December 1975.

[Liskov 79] Barbara A. Liskov and Alan Snyder. "Exception Handling in CLU," IEEE Transactions on Software Engineering. Vol. SE-5, no.

[2]Stdio doesn't tell you what went wrong when an error occurs. We discovered the hard way that the contents of errno are not to be trusted after a stdio error.

6, November 1979.

[Meyer 88] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, 1988.

[Nelson 91] Greg Nelson. Systems Programming with Modula-3. Prentice Hall, 1991.

[Randell 75] Brian Randell. "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering. Vol. SE-1, no. 2, June 1975.

[Roberts 89] Eric S. Roberts. "Implementing Exceptions in C," Research Report 40, Digital Systems Research Center, March 21, 1989.

[Steele 90] Guy L. Steele Jr. Common Lisp. Digital Press, 1990.

[Yemeni 85] Shaula Yemeni and Daniel Berry. "A modular verifiable exception-handling mechanism," Transactions on Programming Languages and Systems. Vol. 7, no. 2, April 1985.

Author Information

After 7 years of hacking UNIX, C, Macintosh, and graphical user interfaces, Doug Moen received his B.I.S. from the University of Waterloo in 1987. His bachelor's thesis was the design of a programming language (Goal) with powerful abstraction mechanisms and a polymorphic type system based on dependent types. In 1988, after graduating, Doug joined Interactive Image Technologies where he made major contributions to an object-oriented, platform independent graphical user interface library, and was the principle designer and architect for HyperCase, a multi-media authoring tool. In 1989, Doug moved to Nixdorf, where he has made substantial contributions to the design and implementation of EMS, a programmers toolkit for building document image processing systems. Doug is now available for employment. Reach him at doug@sni.ca, or at (416) 977-4907, or at 77 Carlton Street #1504, Toronto, Ontario, M5B 2J7.

Appendix: Notify and Signal Conditions

This paper has dealt with 2 kinds of exceptional conditions which can be detected by a library function: faults and failures. Both of these kinds of conditions result in the termination of a function call once they are detected. But there is a third class of exceptional conditions: these are conditions which need not prevent the function from completing its task, but which may nevertheless need to be reported to the caller before the function has completed its task. Goodenough [Goodenough 75] distinguishes two types of exceptional conditions which fall into this category, which he calls *notify* and *signal* conditions. A handler for a *notify* condition is prohibited from terminating the operation. A handler for a *signal* condition is given the choice of terminating the operation, or fixing the problem and resuming it.

For an example of a notify condition, consider pclose(3), which repeatedly calls wait(2) until the process associated with the argument stream has exited. The exit statuses of child processes which are different from the process that pclose() is attempting to wait for are simply thrown away, and programs that call pclose() have no way of obtaining this information. This is a design flaw in pclose() which could be rectified through the use of a mechanism for reporting notify conditions.

For an example of a signal condition, consider a function which is communicating with a remote process over a network connection. If the remote process has apparently stopped talking, then the function has two choices: it can report a failure, or it can keep trying to communicate with its peer, on the assumption that the peer might resume communication in a few seconds. In practice, timeouts are often used in this kind of situation. A better approach might be to report a signal condition, and let the caller attempt to fix the problem, or obtain advice from the user on whether to retry or abort.

A second example of a signal condition is a memory allocator which gives the caller an opportunity to free cached memory blocks before reporting a failure.

As I have tried to show, signal and notify conditions are real. The question is, what combination of language features and programming conventions are most appropriate for dealing with them?

One approach is to incorporate the reporting of signal and notify conditions into the same exception handling mechanism that is used to report failures. This leads to the retry model of exception handling. In this model, when an exception is raised, the context raising the exception is not immediately terminated. Instead, the exception handler is given the choice of terminating the function that raised the exception, or resuming it. The Mesa programming language supported this model of exception handling; so does Common Lisp [Steele 90]; and so does Allman and Been's exception handler for C [Allman 85].

An evident disadvantage of implementing the retry model of exception handling in C is that it is necessary to make exception handlers into separate functions. Not only is this grossly inconvenient, but the exception handlers do not have access to the local variables associated with the statement block that they are associated with.

The retry model of exception handling provides dynamically scoped handlers for signal and notify conditions that are associated with specific blocks of code. It is not clear that this is the best scoping regimen. Alternatives are to associate exception handlers with modules, or to associate them with objects. Handlers with module scope can be implemented by registering a callback function with the appropriate module. Handlers with object scope can be implemented using object oriented programming, by overriding a virtual function in the class of the object.

I have not yet developed a personal philosophy concerning the proper treatment of signal and notify conditions. However, I would like to show how dynamically scoped handlers for these conditions could be added to the EMS error handling system. The technique could probably be adopted to work with any C-based termination-model exception handling system.

The function

```
int notify(Error_t *)
```

is used by a function to notify its client of an abnormal condition; it is used to implement both ''signal'' and ''notify'' exceptions as described above. The Error_t structure is used to describe the abnormal condition. The value returned by notify() is one of the values:

`N_ABORT`	The client wishes the function to abort by raising an exception.
`N_RETRY`	The client wishes the function to keep trying.
`N_IGNORE`	The client ignored the notification.

A client can catch a notification using the following control structure:

```
NOTIFY (fun)
   ... code which may raise
       a notification ...
END_NOTIFY
```

The argument to NOTIFY is a function pointer with type

```
int (*fun) (Error_t *)
```

This function should examine the Error structure (and in particular, the error id), and return N_ABORT, N_RETRY or N_IGNORE. The return value N_IGNORE means that the notification handler function did not recognize this particular notification.

NOTIFY ... END_NOTIFY blocks can be nested in the same way that TRY ... END_TRY blocks can. When a notification is raised, notify() searches the stack of NOTIFY blocks, calling each notification handler function in turn until one of them returns a value different from N_IGNORE. If no handlers are present, or no handler is willing to handle the condition, then notify() returns N_IGNORE.

Regression Testing and Conformance Testing Interactive Programs

Don Libes – National Institute of Standards and Technology

ABSTRACT

Testing interactive programs, by its nature, requires interaction – usually by real people. Such testing is an expensive process and hence rarely done. Some interactive tools can be used non-interactively to a limited extent, and are often tested only this way. Purely interactive programs are rarely tested in any systematic way.

This paper describes testing of interactive line and character-oriented programs via Expect. An immediate use of this is to build a test suite for automating standards conformance of all of the interactive programs in POSIX 1003.2a (interactive shells and tools), something which has not yet been accomplished by any means.

Introduction

Dennis Ritchie said [1] that "A program designed for inputs from people is usually stressed beyond the breaking point by computer-generated inputs." I would add the following: Any program useful to people – interactively – is likely to be useful to programs – non-interactively. A corollary of Ritchie's statement is that correct software function during normal human use is not a very good test of a program's total correctness.

I claim that even when humans are explicitly *testing* interactive software, the results are still quite unreliable. Humans have many drawbacks:

- Humans know what is reasonable, and strive to avoid incorrect input.
- Humans assume programs can do things that have worked in earlier releases.
- Humans get bored quickly, and skip tests.
- Humans forget tests.
- Humans are expensive.

Regression testing requires the same testing to be performed many times. For example, after fixing a bug, a program should be tested without regard to the particular change. Although a modified statement is an obvious place to look for new bugs, subtle bugs can manifest themselves in distant pieces of software. The likelihood of such bugs is low compared to more blatant problems such as incorrect algorithms. Hence, they get short shrift from programmers during testing.

The UNIX tool-building paradigm encourages designing programs that can be used interactively as well as non-interactively. Such programs can be embedded in *pipelines*. Pipelines are sets of programs, where each program produces output that becomes input for the next program in the pipeline. (The first program in a pipeline does not dynamically consume output of another program, but may for example, read a disk. Similarly, the last program does not produce output that is immediately consumed by another process, but may for example, write to a disk or display.) This is the environment in which Ritchie's remark arose.

In practice, there are forms of input that production programs do not generate. For example, programs do not make typing errors and therefore do not (press the backspace or delete key to) delete characters just produced. Similarly, programs do not enter control characters, such as might be used to interrupt a process. This suggests that Ritchie was too optimistic – even computer generated inputs still test only a subset of a program's interface.

Another problem is that some programs are *never* used non-interactively. For example, the UNIX **passwd** program [2] is designed only to be run interactively. **passwd** ignores I/O redirection and cannot be embedded in a pipeline so that input comes from another program or file. It insists on performing all I/O directly with a real user. **passwd** was designed this way for security reasons, but the result is that there is no way to test **passwd** non-interactively. It is ironic that a program so critical to system security has no way of being reliably tested.

Some programs can be run interactively or non-interactively, but detect the difference and modify their behavior accordingly. For example, virtually all programs that prompt when running interactively disable prompting when running non-interactively. Unfortunately, this makes it difficult to automatically test their interactive behavior non-interactively.

Command languages, such as the UNIX shell, offer no way of dealing with programs that "know" they are interacting with a real user. While such languages are rich in control and data structures and can interact with users (prompting and reading responses), they cannot do the same from programs. Command languages in other popular environments such as VMS and DOS are similarly lacking.

Expect – A Tool for Regression Testing Interactions

Expect [3] is a program specifically designed to interact with interactive programs. Expect communicates with processes by interposing itself between them and acting as an intelligent communications switch. Pseudo-ttys [2] are used so that processes believe they are talking to a real user.

This is useful for regression testing interactive programs. Expect reads a script that resembles the dialogue itself. By following the script, Expect knows what can be expected from a program and what the correct responses should be. The script can specify responses by patterns, and can take different actions on different patterns.

Scripts are written in a high-level language (Tcl – Tool Control Language [4][5]) and support:

- **send/expect** sequences – **expect** patterns can include regular expressions.
- high-level language – Control flow (**if/then/else**, **while**, etc.) allows different actions on different inputs, along with procedure definition, built-in expression evaluation, and execution of arbitrary UNIX programs.
- job control – Multiple programs can be controlled at the same time.
- user interaction – Control can be passed from scripted to interactive mode and vice versa at any time. The user can also be treated as an I/O source/sink.

Expect is actually capable of general use in automating or partially automating interactive programs, however this paper will focus on its use in testing.

I will not discuss a high-level test harness. This can be provided by any number of extant packages or shell scripts that are already in use for testing non-interactive programs. This paper focuses on the low-level problems with program interaction itself which differ significantly from non-interactive testing.

Examples and Guidelines

This section of the paper presents guidelines and examples using Expect to test common interactive UNIX tools, building upon earlier work [7]. Familiarity with the rudiments of Expect and UNIX is assumed.

Example – passwd

The UNIX **passwd** program takes a username as an argument, and interactively prompts for a password. The Expect script in Listing 1 takes a username and a password as arguments, and can be run non-interactively.

```
set password [lindex $argv 2]
spawn passwd [lindex $argv 1]
expect "password:"
send "$password\r"
expect "password:"
send "$password\r"
expect eof
```

Listing 1: Non-interactive **passwd** script. First argument is username. Second argument is new password.

In the first line of the script, the variable **password** is set to the value of the expression in brackets. This expression returns the second argument of the script by using the **lindex** command (list **index**). The first argument of **lindex** is a list, from which it retrieves the element corresponding to the position of the second argument. **argv** refers to the arguments of the script, in the same style as the C language **argv**.

The next line starts the **passwd** program, with the username passed as an argument.

In the third line, **expect** looks for the pattern "**password:**". There is no action specified, so the **expect** just waits until the pattern is found before continuing.

After receiving the prompt, the next line sends a password to the current process. The \r indicates a carriage-return. (All the "usual" C conventions are supported.) There are two **send/expect** sequences because **passwd** asks the password to be typed twice as a spelling verification. There is no point to this

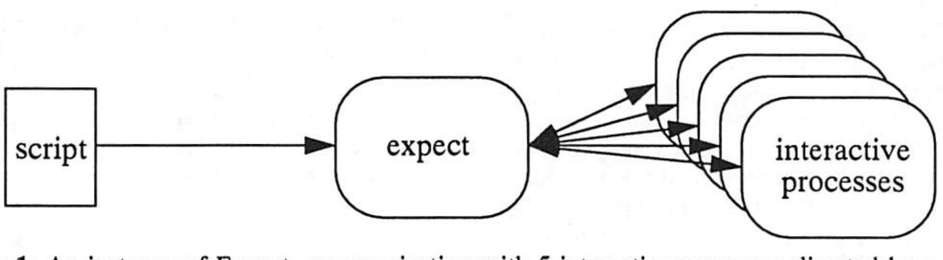


Figure 1: An instance of Expect, communicating with 5 interactive processes directed by a script.

in a non-interactive **passwd**, but the script has to do this because **passwd** assumes it is interacting with a human that does not type consistently.

The final **expect eof** searches for an end-of-file in the output of **passwd** and demonstrates the use of *keyword patterns*. Another one is **timeout**, used to denote the failure of any pattern to match in a given amount of time. Here, **eof** is necessary only because **passwd** is carefully written to check that all of its I/O succeeds, including the final newline produced after the password has been entered a second time.

This script is sufficient to show the basic interaction of the **passwd** command. A more complete script would verify other behaviors. For example, the script in Listing 2 checks several other aspects of the **passwd** program. Complete prompts are checked. Correct handling of garbage input is checked. Process death, unusually slow response, or any other unexpected behavior is also trapped. (The non-interactive functionality of the command is not tested by this script – it is a straightforward task in any language.)

This script exits with a numerical indication of what happened. In this case, 0 indicates **passwd** ran normally, 1 that the user name was bogus, etc. 1X indicates it died unexpectedly and 2X that it locked up, where X is the particular question in **passwd** being checked. Exit numbers are used for simplicity here – descriptive strings could as easily be returned, including messages from the **spawn**ed program itself. In fact, it is typical to save the entire interaction to a file, deleting it only if the command under test behaves as expected. Otherwise the log is available for further examination.

This **passwd** testing script is designed to be driven by another script. This second script reads a file of arguments and expected results. For each set, it calls the first script and then compares the results to the expected results. (Since this task is non-interactive, a regular shell can be used to interpret this second script. As well, it can also be used to test the non-interactive functionality for which **passwd** is responsible, such as checking **/etc/passwd** was correctly updated.) Listing 3 shows a sample data file for testing **passwd**.

```
expect_after {
        eof                                     {exit [expr 10+$question]}
        timeout                                 {exit [expr 20+$question]}
}
set question 0
proc test {args} {
        uplevel {incr question}
        eval [concat expect $args]
}
spawn passwd [lindex $argv 1]
test {
        "No such user"                          {exit 1}
        "New password:"
}
send "[lindex $argv 2]\r"
test {
        "Password too long"                     {exit 2}
        "Password too short"                    {exit 3}
        "Retype new password:"
}
send "[lindex $argv 3]\r"
test {
        "Mismatch - password unchanged"         {exit 4}
        "^\r\n$"
}
test {
        "*"                                     {exit 5}
        eof
}
```

Listing 2: Non-interactive **passwd** script with various tests for behavior at boundary conditions.

The first field names the regression script to be run. The second field is the username. The third and fourth fields are the passwords to be entered when prompted. The last field is the exit value that should match the result of the Expect script. The hyphen is just a placeholder for values that will never be read. In the first test, "`bogus`" is a user name that is invalid, to which **passwd** will respond "`No such user`". Expect will exit the script with a value of 3, which also appears as the last element in the first line of the regression suite data file. In the last test, a control-C is actually sent to the program to see if it aborts gracefully.

In this example, script arguments are sent to programs literally. However, arguments may also be used to name files or otherwise direct scripts.

For example, the following command sends the contents of the file foo to an interactive process.

```
send "[exec cat foo]"
```

The command works as follows. **exec** executes its arguments as an operating system command. On a UNIX system, **cat foo** returns the contents of the file named foo. Unlike **spawn**, **exec** waits for the command to complete and returns the output, which becomes the arguments to **send**, which sends it arguments to the input of the current process.

Example – suspending sleep

The previous script showed an example of sending control characters to a process, which in response simply exited. Some programs actually use control characters as a normal form of input. For example, UNIX shells typically provide a variety of interpretations for control characters such as control-C (kill foreground process), control-Z (suspend foreground process), control-S (stop output), control-D (input end-of-file), control-O (flush output) and others. A shell script containing such characters will not have the desired effect. Indeed, it does not make sense for a shell script to, say, flush output. If this was intended, the script should not have been written to produce the output in the first place.

Most of these control characters are actually first handled by the terminal driver, which then generates a signal handled by special code in the shell. Since no terminal driver is used when a shell script is executed, it is not possible for the script to call this special code upon encountering these control characters. In fact, shell implementors routinely disable all interactive processing as a matter of course. For example, the shell history functions (which enables the user to recall previous commands) are disabled when the shell is running non-interactively. Again, there is no reason for a shell script to ever need this.

The shell is characteristic, therefore, of a class of programs which function differently when run interactively as opposed to non-interactively. There is no way to verify these interactive elements using shell programming.

The only recourse is to take an approach like Expect, which essentially deceives the shell into running as if it were really interactive. An Expect script can send control characters, history commands and any other commands. The script can also manipulate the environment from underneath, for example, by removing the shell's current directory, or killing a child process of the shell, to check its

```
passwd.exp  bogus    -               -          1
passwd.exp  fred     abledabl        abledabl   0
passwd.exp  fred     abcdefghijklm   -          3
passwd.exp  fred     abc             -          2
passwd.exp  fred     foobar          bar        4
passwd.exp  fred     ^C              -          11
```

Listing 3: Example data file for testing **passwd**.

```
spawn csh                 ;# this is a comment
expect "$prompt"          ;# assume prompt is set already
send "sleep 10\r"         ;# run sleep command for 10 secs
exec sleep 1              ;# give time to let sleep begin
send "\cZ"                ;# suspend it
exec sleep 10             ;# wait for 10 seconds
send "fg\r"               ;# let sleep resume
set timeout 5             ;# timeout expect after 5 secs
expect  "*$prompt"        {print "control-Z stopped sleep's clock\n"}
        timeout {print "control-Z didn't stop sleep's clock\n"}
```

Listing 4: Test whether **sleep** counts time while suspended.

response. Listing 4 shows a script which tests whether suspending a **sleep** command actually stops **sleep**'s internal clock.

The script works as follows. A **sleep** is issued for 10 seconds, but is suspended after 1 second. The Expect script than sleeps for 10 seconds, itself, after which it resumes the suspended **sleep**. If Expect then reads a shell prompt, the **sleep** has returned which can only happen if the clock internal to the **sleep** command was still running while it was suspended. If the **sleep** time was indeed suspended, the final **expect** will timeout, since **sleep** will still be running for nine more seconds. (If you run this on most UNIX systems you will find that control-Z does not stop **sleep**'s clock, a counter-intuitive result to most people, but something which must be addressed by implementors and standard writers.)

Example – terminal driver

Scripts can change the default flow of control so that it is not straight-line. Expect supports procedures and the "usual" procedural statements such as **if/then**, **while**, etc. A common use of this is to establish limits during conformance testing. For example, one can write scripts to determine the longest variable name supported in the shell, maximum number of arguments to commands in **ftp**, maximum numbers of messages in **mail** message lists, etc. Using shell scripts to solve this, while possible with some programs, requires the process to be restarted for each test. This can be very expensive for limits that are large. In fact, all of the examples listed here are in the thousands.

An Expect script could generate new tests dynamically using a single process. The overhead in such test generation is extremely low by comparison with multiple process creations.

Listing 5 shows a script that determines the longest input line acceptable to the UNIX terminal driver using the Berkeley line discipline in canonical (i.e., line-oriented) mode.

The script works by writing the letter 'a' in a loop, each time testing that it has been echoed properly. When the buffer fills up, the terminal driver echoes control-G's instead of the typed letter. (On a Sun 4 running SunOS4.1.2, this script reported that the terminal driver only accepted 256 characters, a surprisingly small number.)

```
spawn csh
expect $prompt
for {set i 0} {1} {incr i} {
        send "a"
        expect  {
                "\cG"           break
                timeout         break
                "a"
        }
}
print "driver accepted $i chars\n"
```

Listing 5: Determine longest input line acceptable to terminal driver while in canonical mode.

Example – testing buggy programs

The previous examples were completely automated. However, Expect also accepts input from a real user. It does this in two ways. **send** and **expect** can perform I/O with a real user. In fact, **send** and **expect** can perform I/O with any process that has been **spawn**ed, and the user is just treated as another such process for consistency. A very elegant duality appears here – Expect is a process that plays the part of a user, within which, the user can play the part of a process. The user as process is illustrated by the homunculus in the lower-right corner of Figure 2.

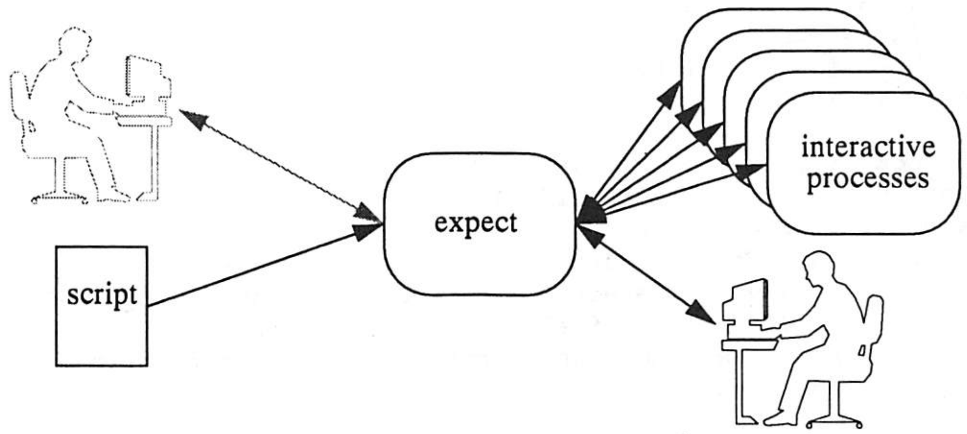


Figure 2: Expect is communicating with 5 processes simultaneously. The script is in control. The user (lower right-hand corner) only sees what the script says to send and is essentially treated as just another process.

The user can take over control from the script and vice versa.

Upon executing the **interact** command, Expect stops reading from the script and creates a direct link between the real user and the process. Thus, it appears to the user as if the process was running interactively in the "usual" way. This is especially convenient when testing a program that takes a large number of interactions before reaching a critical part of the program that is buggy and with which the programmer wants to experiment by hand. Listing 6 shows an invocation of an unnamed application followed by some initialization. In a loop, some interactions occur from a procedure named **punish** (to suggest a difficult set of interactions for the application). Control is then passed to the user, who can now directly interact with the application in an attempt to investigate. This is illustrated by the homunculus in the upper-left hand corner of Figure 2.

When the user presses 'X' (or whatever other escape key is chosen), the user begins speaking directly to the Expect interpreter. The user may enter an Expect command such as **return** (return control to the script), **exit** (exit the script), any valid procedure name, or any valid Tcl command, including even another Expect command or procedure definition. This capability is a great convenience in interactive programs that fail only after a large number of interactions. The user may also run the debugger under Expect, essentially providing the user with a programmable debugger. (Very few debuggers include a general-purpose programming interface that can be applied in this way to interactive programs.)

```
spawn ...
initialize
for {} {1} {} {
        punish          ;# punishing procedure defined elsewhere
        interact X      ;# pass control to user
}
```

Listing 6: Run application through a set of punishing interactions, then let user interact. Repeat indefinitely.

```
spawn csh; set csh1 spawn_id
spawn csh; set csh2 spawn_id
send -i $csh1 "send tty\r"
expect -i $csh1 -re "(/.*)\r"
send -i $csh2 "send write $env(USER) $expect(1,string)\r"
expect -i $csh1 -re "Message from .*"
```

Listing 7: Beginning of a script to start two processes that interact with each other – in this case, via **write**.

```
set csh    [spawn csh]
set cshnew [spawn csh.new]
while {-1!=[gets stdin input]} {
        send -i $csh    $input
        send -i $cshnew $input

        expect -i $csh -re ".+\r\n"
        set output $expect_out(buffer)
        expect -i $csh  $output
        if ![string match output $expect_out(buffer)] {
                send_user "detected discrepancy on input $input\n"
                send_user "original csh output $output\n"
                send_user "new csh output $expect_out(buffer)\n"
                interact
        }
}
```

Listing 8: Run two shells simultaneously from the same input, stopping when there is a difference in their output.

Example – Testing interaction between multiple processes

The previous example alluded to the ability of Expect to control multiple processes. Naturally, this is very important when testing interactions *between* processes.

For example, it might be useful to test the response of a running program to various signals from another process. Expect doesn't need to interactively run programs to generate signals, since it can directly call upon any UNIX command (**kill**, in this case, which is non-interactive). However, something like **write** *does* require two interactive processes to test. Listing 7 displays the beginning of such a script. This script starts two C shells. They may be referred to by their *spawn-id*'s, which are temporarily found in the variable **spawn_id**, set as a side-effect of the **spawn** command. (**spawn**'s return value is the UNIX process id.)

Further commands reference the **spawn_id** by the "**–i**" flag. In this script, shell 1 executes the **tty** command. The result is used by shell 2 when starting a **write** process directed at shell 1. In this script, both processes are run with the same user id, but it is possible to use multiple logins by spawning **login** first.

Notice that this script uses "**-re**" to introduce **egrep**-style regular expressions. While Expect supports both **egrep** and **glob**-style expressions, the **egrep** expressions are much more powerful, and allow very easy access to substrings in matches.

Another use of this multiprocessing ability is to test a new and old program simultaneously until a discrepancy occurs. This is demonstrated in listing 8.

This script reads input from a data file and feeds it to two processes until a difference is found in their output. A more flexible alternative allows the user to drive both programs simultaneously, useful when a user may have difficulty describing a scenario unless actually using and interacting with the program for some time.

Like the UNIX **script** command which records a session, Expect allows interaction to be logged but more flexibly. Listing 9 shows an example.

This script starts two different versions of the same program. In a loop, it listens for output from the programs or the user simultaneously. (The string "`-re .+`" denotes a regular expression of one or more characters.) If the user types, the same keystrokes are sent to both processes. If the programs produce output, it is compared, and if there is a difference, an error message is produced.

In this script, one program's output is arbitrarily selected to copy back to the user. Since the other program's output is just a duplicate, there is no point in copying it also. Similarly, one program's output is copied to a log file. An additional

```
set prog    [spawn prog]
set prognew [spawn prognew]

log_user 0                          ;# turn off default logging
set log [open logfile w]            ;# and log to file explicitly

while {1} {
        expect {
                -i $user_spawn_id -re .+ {
                        send -i $prog    $expect_out(buffer)
                        send -i $prognew $expect_out(buffer)
                        continue -expect
                }
                -i $prog    -re .+ {
                        getoutput prog
                        send_user     $expect_out(buffer)
                        puts $logfile $expect_out(buffer)
                }
                -i $prognew -re .+ {
                        getoutput prognew
                }
        }
        if [mismatch prog prognew] report_error
}
```

Listing 9: Run two programs interactively, let user keystrokes go to both until there is a difference in their output. To avoid confusion, only one program's output is returned to the user.

statement could be added to log user keystrokes, although that is usually not necessary since most programs echo them.

getoutput and **mismatch** are not shown here. **getoutput** simply appends the output to a buffer. The **mismatch** procedure is a tiny bit trickier. It has to account for the fact that programs may produce output at different speeds, perhaps due to kernel scheduling slop. So mismatch just matches to the shorter length of either process's output to the current point, saving anything left over for the next time around.

The technique described here is not limited to two processes. Additional processes may be added, each using one more case in the **expect** statement. **mismatch** itself is designed to take an arbitrary number of arguments.

The script itself can remain the same for varying numbers of processes because Tcl can construct new statements at runtime. In particular, **eval** takes an arbitrary list and executes it as a statement. Thus, a list with the appropriate number of cases can be constructed and evaluated on the fly.

Reality and Guidelines

Using the techniques described in this paper, people have written numerous regression and conformance tests for many interactive programs, such as those of IEEE POSIX 1003.2a. The results have been quite satisfying.

Writing such scripts takes experience, just like any programming task. Generally, however, the hardest part is getting a clear specification of the user interface (UI). The facts of life are, unfortunately, that UIs are notoriously underspecified and nonstandard. However, once a specification is available, translation to an Expect script is straightforward.

To date, only a handful of the simplest interactive commands have had UIs specified by POSIX (much simpler and more boring than the examples here). Test assertions are fairly informal in describing what is permitted, with the understanding that a human will actually be dealing with a program and ''understand'', for example, what a ''prompt'' is.

On the other hand, users *are* automating interactive programs, and explicit UI specifications would help. **ftp** is a good (and bad) example. Each message to the user is preceeded by a number, the idea being that a program can read the message number and discard the remainder of the text line which is meant for a human. In practice, a program has to look at both numbers and the messages themselves. The numbers were never clearly enough specified and each implementation assigns different numbers to differing conditions. Nonetheless, the intent was there and **ftp** has been successfully automated.

Designers of interactive programs should account for the possibility of their programs being automated no matter how hard it is for them to imagine. Some programs say: ''*here is a flag to use when running the program via a script. The flag will change (i.e., simplify) the behavior of the program.*'' This is *not* helpful for testing.

Designers of test assertions should be as detailed as possible. Do not assume that interactive programs will only be run by humans. Even screen-oriented programs such as **emacs** and **vi** can and have been automated.

Users who customize prompts should provide a means for programs like Expect to be able to detect this. For example, a generic shell prompt can be detected by the pattern ''`(%|$|#) `''. In practice, few people leave their prompts unadulterated, and Expect users are encouraged to define a prompt pattern for themselves. For most programs, this is conveniently done in the same initialization file at the same time as the prompt itself is defined. For example, a shell prompt and pattern could be defined in a **.login** file as:

```
set prompt="Yes master (\!%)> "
set prompt_pattern="Yes master (.*)> $"
```

Prompt patterns can be outwitted by similar text in normal program output. This is particularly problematic in a login where a message-of-the-day may contain virtually anything including program examples. A '$' at the end of a pattern (shown above) is helpful, as it allows a match only if nothing more follows.

Performance

Performance is essentially the same as has already been described [3], i.e., excellent. Expect is always faster and more reliable than the alternative – a human. Programs which can be broken by sending control-C or other actions sufficiently fast or oddly timed, can be systematically tested by Expect with different inputs and timing until they break.

As described in [7], Expect recently incorporated a mechanism to slow it down to human-like speeds for more authentic testing. Other parameters are available to control human-like variability characteristics in keyboarding.

Current and Future Work

Expect does not provide explicit support for character-based graphics. In particular, the current implementation understands I/O as strictly stream-oriented. Character-based graphics can be manipulated this way, but the script-writer must be aware of issues such as how graphics are written to the display. Although sufficiently expert coding can simulate this (and indeed, a script exists to play the screen-oriented game of **hunt**), several researchers have experimented with the ability to do screen-oriented interactions. Ideally, a curses inverse is needed to simulate any type of terminal. Researchers are also experimenting with interfaces for describing X or other window system events. These may appear in future releases of Expect.

Applicability

Expect is useful for testing and debugging interactive software. Expect can also be used for building conformance tests of interactive software, such as IEEE POSIX 1003.2a. This paper has presented examples of each of these.

Expect has other uses than program testing. Chief among them is the automation of interactive programs. Nonetheless, Expect has been distributed to over 4000 sites (by request), and the particular use of Expect described herein has proven very popular. Expect has been used to test a wide array of interactive programs, including **tip**, **csh**, many local applications (including Expect itself), and even some non-UNIX applications. While Expect is a UNIX program, it can interact with non-UNIX processes by remotely logging in (e.g., **telnet**, **kermit**) to non-UNIX computers. The language used by Expect does not favor UNIX over any other operating system but is neutral in this regard.

Conclusion

Command shells of UNIX and other common operating systems are incapable of controlling interactive processes. In the past, testing interactive software required a human to press keys and watch for correct responses. After a few iterations, this became quite tiresome. Naturally, people were much less likely to run thorough regression tests after making small changes that they thought *probably* didn't affect other parts of a program.

Expect automates interaction, obviating the need for human effort in regression testing and conformance testing. Using Expect, one can develop automated test suites to assure reliability and consistency with earlier software versions, or conformance with standards, such as POSIX 1003.2a. Expect is also useful for programs that are not yet complete but need interactions in order to evoke failure.

Acknowledgments

This work was supported by the National Institute of Standards and Technology (NIST) Automated Manufacturing Research Facility (AMRF). The AMRF is funded by both NIST and the Navy Manufacturing Technology Program.

Steve Ray, Walter Rowe, Sandy Ressler, Chuck Dinkel, Sheila Frankel, Brian Woodson, and Susan Mulroney provided me with helpful criticism and proofreading of this paper.

I'd like to thank everyone who has recently given me ideas, bug reports and fixes, and porting help, all of which have significantly improved Expect beyond my original ideas and implementation of it. These include John Conti, Steve Summit, Mark Diekhans, Marty Olevitch, Scott Hess, Achille Petrilli, Carl Witty, Stefan Farestam, Jay Shmidgall, John Sellens, Jeff Okamoto, Bob Proulx, Hal Peterson, Wally Strzelec, Ted Gibson, Parag Patel, James Davis, Pete Siemsen, Matthew Freedman, Michael Grant, Phil Shepard, Newson Beebe, Ed Klein, Martin Leisner, Dave Schmitt, Ron Young, Ken Mandelberg, Dongchul Lim, Peter Funk, Karl Lehenbauer, Oliver Kretzschmar, Ian Johnstone, Dave Coombs and, of course, John Ousterhout.

Availability

Since the design and implementation of Expect was paid for by the U.S. government, it is in the public domain. However, the author and NIST would like credit if this program, documentation or portions of them are used. Expect may be **ftp**'d as **pub/expect/expect.shar.Z** from **ftp.cme.nist.gov.** Expect will be mailed to you, if you send the mail message (no subject) **send pub/expect/expect.shar.Z** to **library@durer.cme.nist.gov**.

Disclaimer

Certain commercial products are identified in this article in order to adequately describe projects at NIST. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology.

References

[1] Dennis Ritchie, ''The Evolution of the UNIX Time-Sharing System'', *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Pt. 2, p. 1577, October 1984.

[2] AT&T, *UNIX Programmer's Manual*, Section 8.

[3] Don Libes, ''Expect: Curing Those Uncontrollable Fits of Interaction'', Froceedings of the Summer 1990 USENIX Conference, Anaheim, CA, June 10-15, 1990.

[4] John Ousterhout, ''Tcl: An Embeddable Command Language'', Froceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990.

[5] John Ousterhout, ''*tcl(3) – overview of tool command language facilities*'', unpublished manual page, University of California at Berkeley, January 1990.

[6] Don Libes, ''*The Expect User Manual – programmatic dialogue with interactive programs*'', to appear as a NIST IR, National Institute of Standards and Technology, 1992.

[7] Don Libes, ''Expect: Scripts for Controlling Interactive Processes'', *Computing Systems*, Vol. 4, No. 2, University of California Press Journals, November 1991.

Author Information

Don Libes is the author of ''*Obfuscated C and Other Mysteries*'' and co-author of ''*Life With UNIX*''. In real life, Don is a computer scientist at NIST where his research deals with manufacturing automation. Don hopes one day to automate himself out of a job. This paper describes the first step. Reach him via U.S. Mail at National Institute of Standards and Technology, Bldg 220, Rm A-127, Gaithersburg, MD 20899. His electronic mail address is libes@cme.nist.gov.

NeD: The Network Extensible Debugger

Paul Maybee – Solbourne Computer, Inc.

ABSTRACT

NeD is a debugging server with a programmable network interface. NeD is designed to be flexible and extensible enough to support a wide range of debugging needs. Debugging clients communicate with NeD by sending it programs to execute. The programming language, NeDtcl, is tcl[1] extended with 30 debugging specific functions. NeD can be used as a traditional debugger with a textual interface, but the user would find the language cumbersome. It is designed to be convenient for communication between programs, rather than between program and user. As a demonstration of NeD's viability as a debugging server, the pdb[2] debugger has been retargeted to use NeD as its server.

1.0 Introduction

The nature of UNIX software development constantly changes. Over the past several years workstations with graphics displays have largely replaced ascii terminals, resulting in ubiquitous windowing systems, and the enormous growth in user interface development. The increasing acceptance of object oriented programming has resulted in the widespread use of new programming languages to support it. Multi-process programming has always been common on UNIX machines, but its use has taken on new dimensions with networks and the ability to easily create and communicate with remote processes. Multi-threaded programming is supported now on many vendors' platforms, with more to come in the near future. All of these developments have resulted in a steady increase in the size and complexity of software systems. Yet the scope of changes in debugging technology over the same period of time has been narrow, focusing primarily on user interface issues. Several products, e.g., dbxtool [3] and SaberC (now CodeCenter) [4], have placed window system interfaces over essentially command driven debuggers. Pi [5] and pdb pushed this technology further by completely eliminating the command interface. SaberC made an initial attempt at data structure visualization, and FIELD [6] advanced this technology by the addition of layout methods and iconic structure representations.

FIELD also developed an inter-tool communication system (now available in commercial guise) that allows various other software development and visualization tools to interact directly with the debugger [7]. Tool-interconnection appears to be the next major step in software development environments, with several vendors developing products along the lines of the FIELD work.

Several debuggers now support debugging C++ programs, at various levels. One of the primary reasons for less than complete support in the debuggers has been inadequate compiler generated symbol tables. This problem should gradually disappear as better compilers become available.

The ability to debug multiple processes, even on the same machine, has typically not been supported, and UNIX implementations often have prevented debuggers from even attaching to newly forked process. Large program debugging seems to be an issue that is often ignored in the construction of debugging systems (with the exceptions of pi [12] and pdb). It is common for users go away and come back later when a program has finished loading.

NeD addresses all of these issues. NeD does not have a user interface to speak of; it is a server designed to work with client user interface tools. NeD was built specifically for the support of the C++ language, although it is suitable for use with other languages. However, its capabilities to debug C++ in a completely native mode still suffers somewhat for want of compiler support. NeD supports the debugging of child processes as they fork from debugged parents as well as debugging over the network and debugging multithreaded processes. NeD is extensible; a client may customize NeD's remote interface to more easily and efficiently provide needed services. Thus NeD will continue to be useful in the face of change by allowing clients to expand the power of the debugging server at run time. Finally, because of the use of lazy symbol evaluation, NeD is very efficient, especially in its use of memory.

In the following discussion, "NeD" will refer to the NeD server, "client" will refer to a client of the NeD server, "pdb/NeD" will refer to the version of pdb implemented using NeD, and "subordinate program" will refer to a program being debugged by NeD.

2.0 Features

2.1 Client-Server Debugging

NeD implements the server side of a client/server debugging system. The typical client provides a user interface, the NeD server provides debugging services. NeD executes on the same host as the subordinate program, and uses the operating system provided interface to query and control the process. NeD contains an extended tcl[1] command interpreter. Tcl provides an embedded, list structured, command language that provides "mechanisms for variable, procedures, expressions, etc."; the extension consists of a set of additional embedded functions that provide debugging capability. This extended language is NeDtcl (see Appendix A). The clients execute remote procedure calls that send NeDtcl statements to the interpreter. For example, a call may contain a direct invocation of an embedded debugger function, or contain statements defining new NeDtcl procedures, or it may contain code invoking these previously defined procedures.

The execution of the NeDtcl statements produces results that are forwarded back to the client as the return value of the remote procedure call. In addition , the execution of some statements causes the subordinate to change state, for example to begin execution. When an execution state change occurs NeD generates an event. The event is passed to the NeDtcl interpreter which may handle it, or send it on in a message to the client. NeD generates several classes of events automatically. Clients can also cause additional, user defined, events to be generated. A client handles the event when the message is received, or it can download procedures to the interpreter to catch and process events.

NeD supports a special kind of breakpoint that is placed only on a fork system call. When this breakpoint is hit, NeD modifies the program text following the fork call such that executing the new code will cause the process to suspend. NeD then allows the process to execute the fork. The new child process that is created will immediately suspend. NeD takes control of the parent again, loads the pid of the child, and opens a new socket. NeD then forks. The child copy of NeD attaches to the new subordinate child and accepts connections on the socket. The parent copy of NeD generates a "new_proc" event containing the pid and socket id of the child NeD. At this point the NeD client can connect to the new NeD and debug the child subordinate process (see Figure 1).

NeD clients can extend this picture by attaching NeD servers to multiple processes, even when those processes are on different machines. Pdb/NeD,

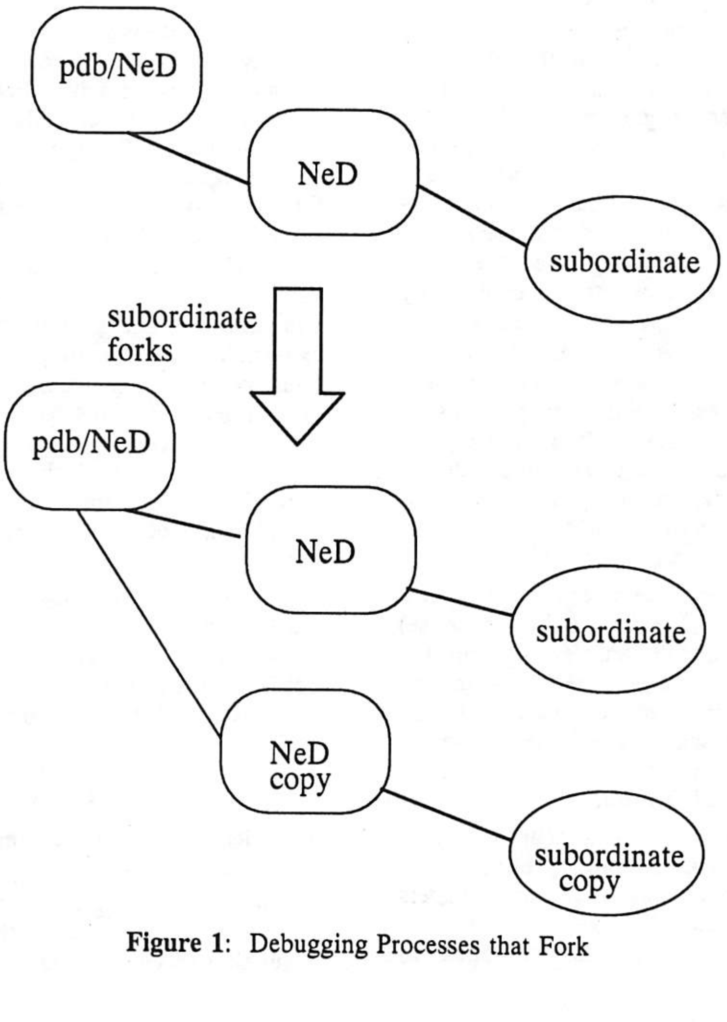


Figure 1: Debugging Processes that Fork

for example, will allow any number of processes to be debugged simultaneously through one set of windows. The user can select the process to view via a menu selection. In addition, a process being debugged will automatically come into view when it hits a breakpoint.

Using a client-server architecture has efficiency drawbacks when more sophisticated tools such as event stream recognizers or program monitors are being used [14,15]. Network latency makes events out-of-date by the time they arrive. Data collection may take up a great deal of network and machine resource. NeD alleviates these problems through its extensiblity. Debugging tools can implement event recognizers or data filters in the server itself (section 2.3) or link additional functionality into the subordinate process(section 2.4).

2.2 Displaying Data

All communication between NeD and its clients is textual, including values returned by expression evaluation requests. NeD supports displaying data structure values through "templates". Templates describe how fields are to be displayed, including conditional display based upon the program state. If an integer variable "i" with a current value of 10 is evaluated the result returned by NeD would be "SIMPLE NUMBER 10", indicating that a simple value was returned. A template for this expression would have a form similar to this result. The template "SIMPLE {hex tim}" would direct NeD to return the value in hexidecimal and as a system time value. Also allowed are octal, decimal, binary, ascii, unsigned decimal, floating, and special formats. The special option allows the user to customize the display of values in a way that makes sense in the context of the subordinate program. This option requires that the name of a function to be found in the subordinate be supplied, e.g., "SIMPLE {hex {spl *myfunction*}}". When this value is to be displayed, myfunction is invoked with the value as input. It returns a pointer to a static text string representing the value.

For aggregate values the format matches that of the data structure. Given the following structure definition:

```
struct s {
     int i;
     struct s *ptr;
};
```

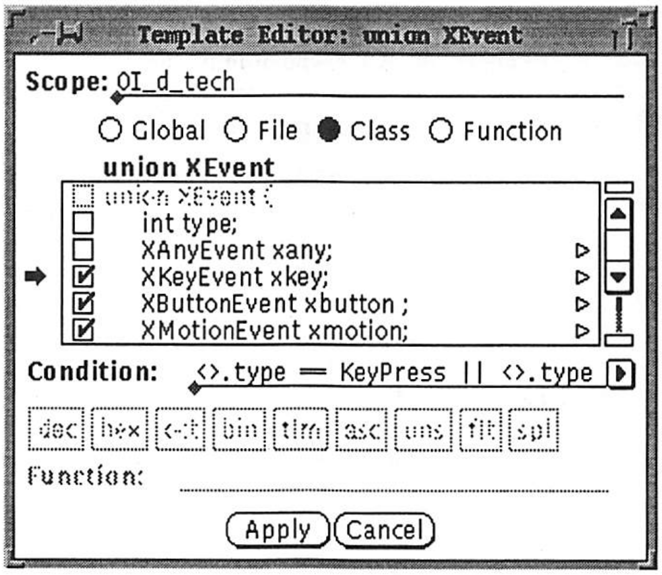
when a variable of this type is evaluated NeD returns:

```
AGGREGATE s {
     {i {SIMPLE NUMBER 10}}
     {ptr {SIMPLE PTR 0xf7fffb00}}
}
```

Each field of an aggregate template contains a three element list:: first the field name, then a conditional expression, and then the field's template. The template that matches "struct s", that displays "i" as above, and displays "ptr" only when "i" is non-zero looks like:

```
set_template {unsigned int} {CLASS foo} {SIMPLE oct}
```

Figure 2: Displaying unsigned integers in class "foo"



Scope: OI_d_tech

○ Global ○ File [✓ ● Class] ○ Function

union XEvent

```
☐ union XEvent {
☐      int type;
☐      XAnyEvent xany;
☑      XKeyEvent xkey;
☑      XButtonEvent xbutton ;
☑      XMotionEvent xmotion;
```

Condition: <>.type == KeyPress || <>.type

Figure 3: Pdb/NeD template editor

```
AGGREGATE s {
    {i {} {SIMPLE {hex tim}}}
    {ptr {<>.i != 0} {SIMPLE hex}}
}
```

Any boolean expression that can be evaluated in the context of the original expression can be used as an evaluation condition. The angle bracket pair (<>) in the example condition refers to the expression being evaluated, "exp", the brackets will be replaced by "(exp)" before evaluation. This is especially useful when displaying union types with tag fields. Each variant of the union can carry a condition such as "tag == mytag". Then when the union is evaluated only the correct union variant appears.

A template can be specified on an individual expression or for all expressions of a given type (in various scopes). For example, all unsigned integers in the scope of class "foo" can be displayed in octal by executing the NeDtcl command shown in Figure 2. Variables of type "unsigned int" evaluated in locations other than member functions of class "foo" will not be affected. Scope designations can include GLOBAL, FILE, CLASS, or FUNCTION scopes. When an expression is evaluated the templates will be searched for a matching type in the inner most matching scope. If no custom template is found then a default global template is used.

The set of custom templates that are in effect can be retrieved from NeD and saved. NeD can then be reloaded with these the next time that the program is being debugged. Templates for entire classes of debugging can be prepared a priori and used by groups of programmers. For example, the X window system [5] defines a union called an XEvent with thirty-three variants. Thirty-one of the variants are the different X event types and one of the variants is the tag field (i.e., the tag is an overlay of the first field in each of the events), the remaining variant is padding. A NeDtcl initialization file could be prepared for X programmers that would include a template for the XEvent; displaying the proper variant depending upon the tag field. Figure 3 shows the pdb/NeD user interface for editing type templates

2.3 Extensibility

NeDtcl's embedded functions provide, more or less, traditional debugger functionality. There are functions that, for example, set and delete breakpoints, advance execution, and evaluate expressions. The interface is stateless. There is no concept of a current function, or current file, as a more traditional debugger would supply. Commands tend to take additional parameters with which to specify program locations or options. For example, in dbx the user can type "stop at 10" to set a breakpoint at line 10 of the current file. When using NeDtcl this operation requires entering code as shown in Figure 4. This makes the language unacceptably cumbersome for human use, but not for computer use. The NeD interface is designed for use by a NeD client that presents an alternate, most likely graphical, user interface. This is not to say that NeD cannot be used as a textual debugger, only that the interface, by default, does not support it well. Since NeDtcl is a complete programming language a "dbx-like" user interface could be written in NeDtcl and loaded into the interpreter. In fact, to facilitate testing NeD, this was done.

NeDtcl does not contain functions that are not directly debugger related (e.g., dbx's "make") or functions that produce information that can otherwise be computed (e.g., dbx's "where"). Non-debugging functionality, such as recompiling objects, is

```
set_break [addr_of {thisfile.c 10 {}}] {} {} stop
```

Figure 4: A breakpoint in NeDtcl

```
### definition of traceback(stack_num, max_depth)

proc traceback {{stack_num 0} {max_frames 100}} {         ### compute traceback
    set d [min $max_frames [depth stack_num]]             ### how many frames?
    set trace ""                                          ### initialize list
    for {set i 0} {$i < $d} {set i [expr {$i + 1}]} {    ### for each frame
        set pc [get_pc $stack_num $i]                     ### get the pc
        set loc [location_of $pc]                         ### get source location
        ### concat the pc, location, and evaluated parameter list
        lappend trace [list $pc $loc
                [eval_list $loc $stack_num $i
                    [list_of_params $loc]]]
    }
    return $trace                                         ### return completed list
}
```

Figure 5: A traceback

assumed to be provided, if necessary, by the NeD client. Complex functionality, such as displaying a stack traceback, can be loaded into the interpreter. In the pdb/NeD implementation, the first operation pdb directs NeD to do at start-up is load a file of NeDtcl procedure definitions into the server's interpreter; one of which is "traceback"; see Figure 5.

The information returned by *traceback* is a list with the same information as is contained in a stack traceback obtainable from any source level debugger. *Traceback* traverses the execution stack returning a description of each stack frame. Each description is a list containing the pc, source location (including function name), and parameter list for the frame. A parameter list contains the name and current value of each parameter. In these examples itialics are used for names (e.g., *traceback*, *get_pc*, *eval_list*, and *list_of_params*) defined in the pdb/NeD start-up file; names in bold (e.g., **depth** and **location_of**) are embedded NeDtcl functions. The remainder is tcl code.

Getting a traceback requires only one remote procedure call. The previous version of pdb required one procedure call to get the location and parameter names and then another call for the evaluation of each parameter. An alternate way to decrease the remote traffic would have been to modify the former interface to include all the information returned by this traceback function. However, the problem with a fixed interface is that it does not provide enough flexibility. It may be that a future graphical debugger interface would require much less, or much more information for each stack frame (e.g., no parameter evaluation or values for all local variables). NeD provides this flexibility.

NeDtcl also provides support for the interception and generation of NeD events. Figure 6 implements a conditional step function. The procedure *step_until1* takes an expression (and some context information) and steps the program one source line at a time until the expression evaluates to true.

Figure 6 implements a function similar to a "watchpoint", i.e., break execution on a memory location content change rather than on a memory location being executed. At each execution of **step** the process changes state twice; it starts executing and then it stops executing. Each of these state changes will result in an event being sent to the client. If the client normally processes these events to update a display then this may result is a great deal of unwanted screen painting and network traffic. This procedure can be easily modified to intercept the events.

When NeD generates an event it invokes the NeDtcl function *process_events*. *Process_events* cycles through a list of event processor functions looking for one that has expressed an interest in the event. If one is not found, the event is sent to the client via the **event_pass** function. So to intercept the process state change messages, the "step until" procedure must be modified to add an event processor. The function *on_event* is used to post or remove an event processor. The first parameter indicates the event class to catch, the second parameter is a command to execute. If the command returns non-zero then the event has been handled. *Step_until2* installs a simple handler for all "proc_state" events before invoking *step_until1*, and then removes it afterwards. *Step_until2* also sends a "proc_state running" event to the client before

```
### step_until1(location stacknum expression)
###                 : no event interception

proc step_until1 {loc stacknum exp} {
    while {![proc_eval $loc $stacknum 0 $exp {} 0]} {
      step 0 {}
    }
}
```

Figure 6: A watchpoint

```
### step_until2(location stacknum expression)

proc step_until2 {loc stacknum exp} {
    event_pass {proc_state running}          ### program set running
    on_event proc_state {return 1}           ### handle event
    step_until1 $loc $stacknum $exp
    on_event proc_state                      ### remove handler
    checkstate          1                    ### generate final event
}
```

Figure 7: A more complex example

beginning execution and causes NeD to generate a "proc_state" event after stepping has completed. This allows the client's normal event processing to see the "step until" operation as it would a single step; see Figure 7.

The previous examples demonstrate NeD's ability to display the state and control the execution of a subordinate process, the following examples show that NeD can also perform non-trivial data queries on subordinates. Procedure *list_length1* will calculate the length of a linked list. It assumes that the list structure contains a next field, by default named "next", pointing to a structure of the same type. Given the head of the list, the function iterates until it discovers a next field with a special value, by default 0x0. Each succeeding next value is computed by adding ".next" onto the previous expression and reevaluating. See Figure 8 for the listing.

This implementation will work, but the size of the expression being evaluated grows with the length of the list, making it a less than perfect solution. An alternate implementation makes use of the expression evaluator's "typeof" intrinsic function to keep the length of the expression constant. Instead of using the previous expression to get the next expression to evaluate, *list_length2* uses the value of the previous expression. The next expression is generated by casting the value to a pointer to the type of the structure and then adding "->next". See Figure 9.

Queries such as this can be very useful in building advanced graphical user interfaces, for instance structure browsers. Network latency can be reduced by filtering data at the server rather than requiring network transmission. As with the templates, groups of users working on similar problems can share modifications to enhance a project's debugging environment.

2.4 Subordinate Extensibility

NeD allows the subordinate program to be extended at run time. NeD will load dynamic libraries into the executing subordinate and then allow functions in the library to be invoked through the debugger interface. This can be useful for supporting debugging aids, such as tools that traverse program data structures looking for inconsistencies, or performance monitoring functions. For example, the list_length function of section 2.3, or a free space analysis routine could be supplied by the user in a shared library. One of the first uses this was put to at Solbourne was to provide a file descriptor status utility. A function that 'stat's all the processes file descriptors is linked in and invoked. The function dumps a report to a file; the utility reads the report and displays the information to the user.

Newly linked library code can also be directly called by the subordinate process by inserting calls into the program text. This feature provides the facilities needed for less intrusive debugging and monitoring tools, such as Parasight [13]. The Parasight paradigm involves inserting new functions into a running process to collect performance and trace data. It is orders of magnitude faster to have the program collect its own data, than to have the debugger stop it continually to do so.

NeD causes the subordinate process to execute dynamic linker calls in order to load new libraries. Under SunOS, dynamic linker routines are always resident with any dynamically linked process. NeD exposes the 4 dynamic library routines necessary

```
### list_length1(loc stacknum head field lend): length of list

proc list_length1 {loc stacknum head {field next} {lend 0x0}} {
    set next [simplevalof [proc_eval $loc $stacknum 0
    $head {} 0]] set exp $head for {set i 0} {$next != $lend} {set i [expr
    $i+1]} {
      set exp ($exp).$field set next [simplevalof
      [proc_eval $loc $stacknum 0 $exp {} 0]] } return $i }
```

Figure 8: Calculating length of a list

```
### list_length2(loc stacknum head type field lend): length of list

proc list_length2 {loc stacknum head {field next} {lend 0x0}} {
    set next [simplevalof [proc_eval $loc $stacknum 0
    $head {} 0]] set type [simplevalof [proc_eval $loc
    $stacknum 0 typeof($head) {} 0]] for {set i 0} {$next!= $lend} {set i
    [expr $i+1]} {
      set exp (($type *)$next)->$field set next [simplevalof
      [proc_eval $loc $stacknum 0 $exp {} 0]] } return $i }
```

Figure 9: An alternate list length calculator

(dlopen, dlclose, dlsym, and dlerror) by placing symbols for them into the symbol table. Expressions can then reference the symbols to access the functionality they provide. On systems where the dynamic libraries are not always present in the subordinate the debugger must either use an alternate mechanism or require the image to be linked specially. Figure 10 is a sample NeDtcl function that invokes an arbitrary function in an arbitrary shared library.

3.0 Implementation

NeD is more than a demonstration of concept; it will form the basis for the next generation of Solbourne's debugger products. As such, it must be efficient and provide a full range of debugging features. NeD is still under development but is already robust enough and functional enough to support the pdb debugger. NeD is implemented for the SPARC architecture under SunOS 4.1.1. It can

```
proc dyncall {library func args} {
    set l simplevalueof
        [proc_eval {{} 0 {}} 0 0 dlopen($library,1) {} 0]
    set f simplevalueof
        [proc_eval {{} 0 {}} 0 0 dlsym($l, $func) {} 0]
    proc_eval {{} 0 {}} 0 0 ((int ())$f)($args) {} 0
    proc_eval {{} 0 {}} 0 0 dlclose($l) {} 0
}
```

Figure 10: An arbitrary function evaluator

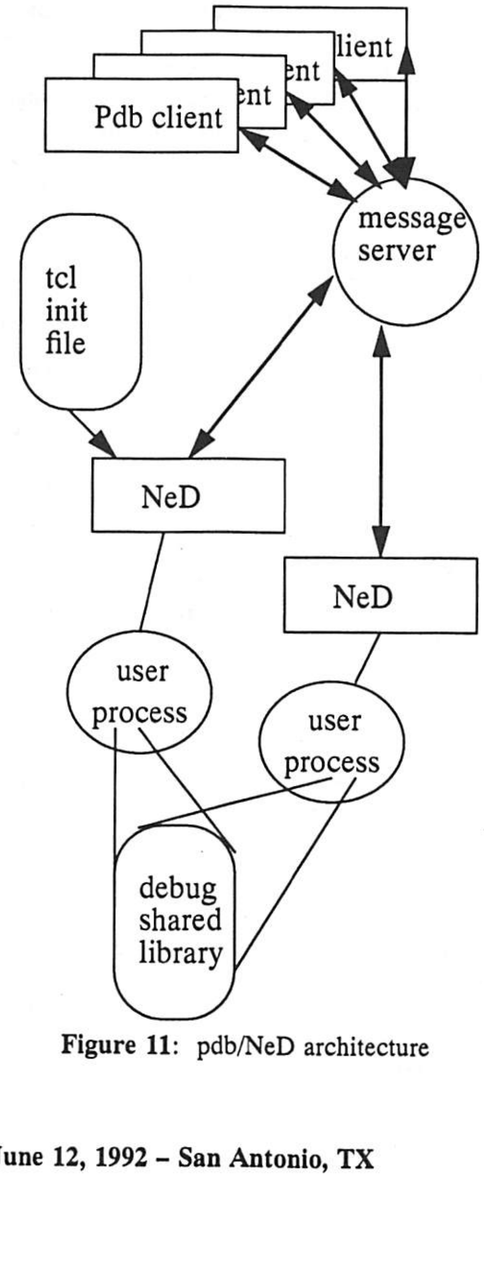


Figure 11: pdb/NeD architecture

debug C, C++, and Fortran programs at the source level, understanding proper target language syntax. Initial tests indicate that NeD uses considerably fewer resources than conventional debuggers such as dbx and gdb. Table 1 shows a comparison of the time and memory required to load a C++ program containing approximately 144,000 symbols. The tests generating these results were run on a Solbourne S4000 with 40 meg of memory.

resource	NeD	gdb	dbx
real time (sec)	33	29	140
memory (megs)	4	16	39

Table 1: Comparison of NeD, gdb, and dbx

4.0 Futures

4.1 Pdb/NeD Architecture

Pdb/NeD clients currently communicates directly with NeD servers. However they will eventually communicate through a FIELD-like message server [7,9]. Pdb/NeD is currently a single client, and that too will change. The pdb user will be able to invoke multiple, independent clients that communicate through the message server. Clients will be specialized, for example to display and edit annotated source, to evaluate expressions, to graph data structures, or to monitor communications. NeD servers will respond to queries from any client. Clients will extend the servers by downloading procedure definitions, or by having the servers load initialization files, or by mapping shared libraries that provide additional functionality into the subordinate's address space (see Figure 11).

4.2 Multi-threaded Process Debugging

NeD contains support for debugging multi-threaded programs, but the current implementation base (i.e., SunOS 4.1.1) does not implement multi-threaded processes, thus there is never more than one thread per process. NeD will be ported to a multi-threaded version of the SVR4 operating system in the near future. The port will also result in a switch from using the ptrace debugging interface to using the /proc interface [10]. A benefit of the move to the /proc interface is that it greatly simplifies debugging forked processes.

4.3 Grafting NeDtcl to Other Debuggers

A debugger for a modern software development environment cannot succeed without associated tools providing graphical user interfaces. In fact, most debuggers offer very similar underlying capabilities, with the main distinction between them coming in the interface or higher level analysis capabilities. User interface software written for the X window system, in our experience, has been easy to port between platforms. Porting debuggers is a much more difficult task since the debugger is dependent upon the operating system's debugging interface, the compiler-debugger interface, and machine architecture. Projects to standardize UNIX and to standardize the symbols generated by compilers, e.g., DWARF [11], will help make the task easier if adopted. But the task will not become as easy as porting X library dependent code, or other application software, in the foreseeable future.

The NeD viewpoint is that the investment in graphical tools can be most easily capitalized on if the debugger doesn't have to be ported. If the graphical tools are written to use a defined debugger interface such as NeDtcl, then only the interface must be ported to a new debugger. There are debuggers available, e.g., the Free Software Foundation's gdb, that have already been ported to many platforms. If such a debugger were to understand the NeDtcl interface then new tools could more rapidly be available across different platforms.

The NeDtcl portion of the NeD server amounts to about 1600 lines of C++ code, most of which is a straightforward translation of the parameters to, and results generated by, the embedded functions from character to internal format. The special formats required for expression value output (see "value" in Appendix A, Data Types) is not such a large change to the traditional value display procedure in any debugger. This would typically be a recursive routine that traversed an internal structure and prints out the contents of each field. The modifications consist mainly of printing some additional brackets and typing information that must already be available to such a module.

5.0 Acknowledgments and Availability

Andrew Gerber implemented the graphical user interface for NeD, making sophisticated testing possible. The members of the OI group served as guinea pigs for early versions of pdb/NeD. Their suggestions and comments have helped enormously.

NeD is still under development but will be available from Solbourne Computer's software business unit later this year. The initial version will be targeted for SPARC compatible computers running UNIX SVR4.

References

[1] Ousterhout, J. K. *Tcl: An Embeddable Command Language*, Proceedings of the Winter 1990 USENIX Conference, Washington D. C. (January 22-26, 1990)

[2] Maybee, P. *pdb: A Network Oriented Symbolic Debugger*, Proceedings of the Winter 1990 USENIX Conference, Washington D. C. (January 22-26, 1990)

[3] Adams, E. and Muchnick, S. S. *Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations*, Software - Practice and Experience, V16 N7, July 1986, pp.653-669

[4] Kafer, S. Lopex, R. and Pratap S. *Saber-C An Interpreter-based Programming Environment for the C Language*, Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA (June 20-24, 1988)

[5] Cargill, T. A. *The Feel of Pi*, Proceedings of the Winter 1986 USENIX Conference, Denver, CO (January 15-17, 1986)

[6] Reiss, S. P. *Interacting with the FIELD Environment*, Brown University Department of Computer Science Technical Report CS-89-51 (May, 1989)

[7] Reiss, S. P. *Integration Mechanisms in the FIELD Environment*, Brown University Department of Computer Science Technical Report CS-88-18 (October, 1988)

[8] Scheifler, R. W., Gettys, J., and Newman, R. *X Window System C Library and Protocol Reference*, Digital Press, 1988

[9] *ToolTalk Programmers Guide*, SunSoft, Inc. Revision A of September 1991

[10] Faulkner, R., and Gomes, R. *The Process File System and Process Model in UNIX System V*, Proceedings of the Winter 1991 USENIX Conference, Dallas, TX (Jan. 21-25,1991)

[11] *DWARF Debugging Information Format*, UNIX International Programming Languages Special Interest Group, October 21, 1991, DRAFT

[12] Cargill, T. A. *Pi: A Case Study in Object-Oriented Programming*, C++ Workshop Proceedings, Santa Fe, NM, USENIX Assoc. (Nov 9-10, 1987)

[13] Aral, Z. and Gertner, I., *High-Level Debugging in Parasight*, Proceeding of Workshop on Parallel and Distributed Debugging, published as SIGPLAN Notices, V24, No. 1, pp151-162, January 1989.

[14] Bates, P., *Debugging Heterogeneous Distributed Systems Using Event-BAsed Models of Behavior*, Proceeding of Workshop on Parallel and Distributed Debugging, May 5-6, 1988, published as ACM SIGPLAN Notices, V24, No. 1, pp 11-22, January 1989.

[15] Spezialetti, M. *An Approach to Reducing Delays in Recognizing Distributed Event Occurrences*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 20-21, 1991, published as ACM SIGPLAN Notices, V26, No. 12, December 1991.

Author Information

Paul Maybee is a software engineer with Solbourne Computer, Inc. He is the principle designer of Solbourne's debugger products including pdb and NeD. Prior to joining Solbourne, Paul attended the University of Colorado where he received a Masters degree in Computer Science. Reach him via U.S. Mail at Solbourne Computer, Inc., 1900 Pike Rd. Longmont, CO 80501, or electronically at paulm@solbourne.com.

Appendix A: NeDtcl grammar

This grammar defines the NeDtcl extensions to tcl. All non-chain rule productions correspond to tcl lists. Thus a production with n terms on the right hand side refers to a tcl list of length n (unless n = 1). Terminals appear in uppercase, non-terminals in lowercase. All non-terminals ending in "_name" are chained to STRING, these productions do not appear. Functions can complete successfully or fail. Functions are listed with the parameters types each expects. Each function is followed, optionally, by the type of value each returns when it completes successfully and the type of event it generates.

Embedded Functions

```
command = proc_cmd | scope_cmd | obj_cmd | expr_cmd | sig_cmd | inst_cmd
                  | exec_cmd | info_cmd

proc_cmd =                                  ### process/stack commands
        "stacks"                            # returns stackid_list
|       "depth" stackid                     # returns INT
|       "location_of" ADDRESS               # returns location
|       "addr_of" location                  # returns ADDRESS

scope_cmd =                                 ### scope query commands
        "list_of_variables" context         # returns var_list
|       "list_of_types" context             # returns type_list
|       "list_of_funcs" context             # returns func_list
|       "list_of_files"                     # returns file_list
|       "list_of_classes"                   # returns class_list

obj_cmd =                                   ### object command
                "load_obj" file_name        # generates object_load

expr_cmd =                                  ### expression evaluation commands
        "proc_eval" location stackid stackdepth expression template async
                                            # returns value, generates expr_eval
|       "set_template" type_name context template # returns BOOLEAN
|       "get_template" type_name context    # returns template

sig_cmd =                                   ### signal processing commands
        "handle" signal_name sighow         #
|       "signal" signal_name                #
|       "sigstate"                          # returns signal_list

inst_cmd =                                  ### instrumentation commands
        "set_break" ADDRESS condition expression break_action
                                            # returns break_id
|       "set_watch" ADDRESS size condition expression break_action
                                            # returns break_id
|       "catchfork"                         # returns break_id
|       "delete_break" break_id             #
|       "get_breaks"                        # returns break_list

exec_cmd =                                  ### process execution commands
        "attach" pid                        #
|       "coreattach" corefile_name objectfile_name# returns INT
|       "detach"                            #
|       [ "Stepi" | "Nexti" | "Step" | "Next" | "Cont" | "Finish" ]
                        stackid signal_name# generates proc_state

info_cmd =                                  ### informational commands
        "config" config_kind                # returns config_list
|       "checkstate" async                  # returns proc_desc,
                                            # generates proc_state
|       "event_pass" event                  #
|       "interp" interp_kind address n_a n_b  # returns interp_list
|       "set_param"      param_name      param_value
```

Events

```
event = object_load | expr_evaluated | proc_state | trace

object_load = "objload" time                 ### object loading is complete
expr_evaluated =                             ### expression eval is complete
        "expreval" location stackid stackdepth expr_value type_name
proc_state =                                 ### process state change
        [ "stopped" | "running" | "terminated" | "unattached" ]
new_process = "newproc" pid host_name port ### new process to debug
trace =                                      ### tracepoint executed
        "tracemsg" location stackid stackdepth expr_value type_name
```

Data Types

```
aggregate_temp = "AGGREGATE" field_temp      ### struct/union template
aggregate_value =                            ### struct/union expression value
        "AGGREGATE" agg_name field_value*
async = BOOLEAN                              ### asynchronous notification flag
break_action = "STOP" | "GO"                 ### stop==breakpoint, go==tracepoint
break_desc =                                 ### breakpoint description
        [       "BREAKPOINT" normal_break_desc |
                "WATCHPOINT" watch_desc |
                "FORKPOINT" ] break_id
break_id = INT                               ### instrumentation identifier
break_list = break_desc*                     ### breakpoint description list
class_list = class_name*                     ### class name list
condition = expression                       ### conditional expression
config_kind = "registers" | "signals"        ### configuration types
context =                                    ### scope description
        scope_type scope_name | "LOCATION" location
expr_value = expression value                ### result of evaluating an expression
expression = STRING                          ### C/C++/Fortran expression
field_temp =                                 ### struct/union field template
        field_name print_condition template
field_value = field_name value               ### struct/union field expr value
file_list = source_file*                     ### list of source files
func_list =                                  ### list of functions in source
        function_name file_name path_name line_num ADDRESS
interp_desc = intruction_desc                ### memory interpretation
interp_kind = "INSTR"                        ### how memory should be interpreted
interp_list = interp_desc*                   ### list of memory interpretations
instruction_desc =                           ### instruction description
        location instr_tranlation symbolic_name
instr_tranlation = STRING                    ### instruction disassembly
line_number = INT                            ### source line number
location =                                   ### source location
        file_name line_number function_name
n_a = INT                                    ### number of translations
                                             ### after address
n_b = INT                                    ### number of translations
                                             ### before address
normal_break_desc =                          ### breakpoint description
        ADDRESS condition expression break_action
pid = INT                                    ### UNIX process id
port = INT                                   ### inet port number
print_condition = expression                 ### conditional expression for
                                             ### templates
print_kind =                                 ### simple print formats
        "dec" | "hex" | "oct" | "uns" | "bin" |
        "tim" | "asc" | "ins" | "flt" | "spe"
proc_desc = STRING                           ### description of process being
```

```
regclass_list = register_class_name*       ### list of register classes
regname_list = register_name*              ### list of register names
sig_state = signal_name sighow             ### how signal will be handled
sighow = "IGNORE" | "CATCH"                ### signal handling types
signal_list = sig_state*                   ### list of signal handlings
simple_temp = "SIMPLE" print_kind*         ### simple type template
simple_value =                  "SIMPLE"   ### simple type value
        ["NUMBER" | "PTR" | "MSG" | "ERROR" |
        "VOID" | "ENUM" | "COMPLEX" | "STRING"] STRING
size = INT                                 ### number of bytes
scope_type =                               ### kinds of scopes
        ["GLOBAL" | "CLASS" | "FILE" | "FUNCTION" ]
source_file = file_name path_name          ### source file description
stackdepth = INT                           ### number of frames on call stack
stackid = INT                              ### identifier for call stack
storage_class =                            ### variable storage classes
                "local" | "global" | "static" | "param" | "refparam" |
                "resparam" | "register" | "regparam" | "unknown" |
                "classmem" | "common"
template = simple_temp | aggregate_temp    ### value print templates
time = INT                                 ### object file time stamp
type_list = type_name*                     ### list of program types
value = simple_value | aggregate_value     ### expression value
var_list = variable*                       ### list of program variables
variable = variable_name storage_class     ### variable name and class
watch_desc =                               ### description of a watchpoint
        ADDRESS size condition expression break_action
```

The Continuous Media File System

David P. Anderson – University of California, Berkeley
Yoshitomo Osawa – Sony Corporation
Ramesh Govindan – University of California, Berkeley

ABSTRACT

Handling digital audio and video data (''continuous media'') in a general-purpose file system can lead to performance problems. File systems typically optimize overall average performance, while many audio/video applications need guaranteed worst-case performance. These guarantees cannot be provided by fast hardware alone; we must also consider the interrelated software issues of file layout on disk, disk scheduling, buffer space management, and admission control. The Continuous Media File System (CMFS) is a prototype file system that addresses these issues.

Introduction

Support for digital audio/video (continuous media, or CM) has emerged as an important area in computer system design. CM capabilities will greatly expand the role of computer systems and will enrich the user interfaces of existing applications. Much effort is being directed toward ''integrating'' CM, that is, handling CM data in the same hardware and software framework as other data. This approach provides advantages in flexibility and generality; however, it introduces performance problems because CM traffic contends for hardware resources (CPU, disk, network) with other traffic.

Hardware speedup alone cannot solve these performance problems. It is necessary to schedule resources or limit workload (or both) in a way that reflects the performance requirements of CM traffic. For example, applications might vary data rates (e.g., reducing video resolution or frame rate) in response to changing load conditions. Alternatively, the system might allow applications to ''reserve'' resource capacity; resources must then be scheduled accordingly. The ability to reserve capacity is especially important for file systems, since in general the data rate of stored CM data is fixed.

CMFS (Continuous Media File System) is an experimental disk storage system for integrated CM. CMFS has the following properties:

- Clients of CMFS can reserve capacity in the form of *sessions*, each of which sequentially reads from or writes to a file with a guaranteed data rate.
- Multiple sessions, perhaps with different data rates, can exist concurrently, sharing a single disk drive.
- Non-real-time traffic is handled concurrently. Thus CMFS can be used as a general-purpose file system that handles CM data as well.

To provide these capabilities, CMFS addresses several interrelated design issues: disk layout, admission control (acceptance or rejection of new sessions) and disk scheduling. CMFS does not address high-level issues such as security, naming and indexing, or document structuring; these are left to higher levels [9, 10]. CMFS simply provides the ability to source or sink byte streams to/from storage at guaranteed rates. CMFS is influenced by several previous CM file systems [1, 5, 8, 11]. However, CMFS is more general than these systems: it supports read and write sessions, variable-rate files, and multiple sessions with different rates. It also supports non-real-time traffic more effectively.

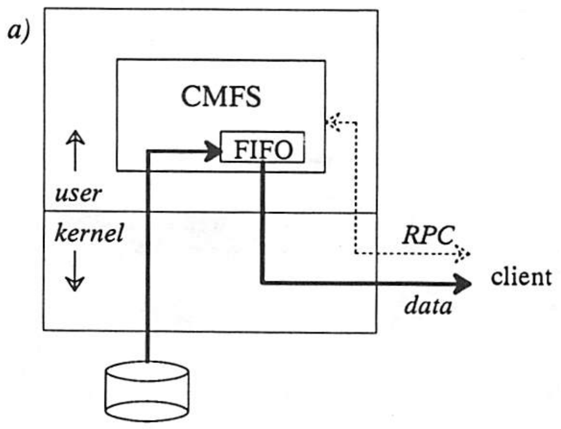
The Client Interface to CMFS

What exactly is meant by ''guaranteed data rate''? It is neither feasible nor desirable that CMFS should deliver data at a completely uniform rate, one byte every X seconds. Ideally, the semantics of a CMFS session should accommodate variable-rate files, work-ahead, and client pause/resume. We have developed a semantics that handles these cases in a simple and uniform way. Our semantics also provides a duality between reading and writing, thus simplifying CMFS.

To precisely describe the semantics of a CMFS session, we need a model for how CMFS interacts with its clients. Our model is as follows. Each session has a main-memory FIFO buffer for data transfer between CMFS and the client. For a read session, CMFS appends data to the FIFO and the client removes data (blocking when the FIFO is empty). For a write session, the client appends data (blocking when the FIFO is full) and CMFS removes it. Performance guarantees are defined entirely in terms of data insertion in, and removal from, the FIFO.

Further details are intentionally left unspecified, since the model can be realized in various ways. For example, a CM-capable file system may run in the OS kernel or in a protected user-mode virtual address space (see Figure 1). It may communicate with clients via traps (system calls) or via RPC; the

FIFO may be a memory-mapped stream [6] or a kernel data structure accessed by `read()` system calls.



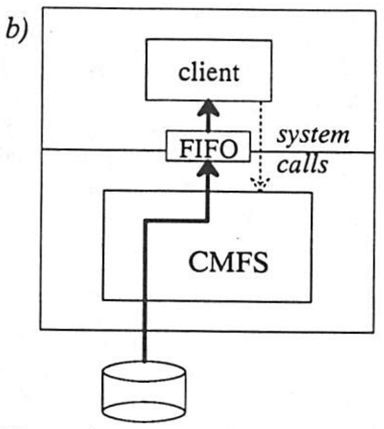


Figure 1: A CM-capable file system can run at the user level, communicating with its clients over network connections (a). Alternatively, it can be implemented in an OS kernel, with client data access by a shared-memory FIFO (b).

The CMFS prototype is implemented as a user-level UNIX process (Figure 1a), and clients communicate data via flow-controlled network connections. Data is removed from the FIFO of a read session whenever the corresponding network connection is ready to accept data.

The Semantics of a Session

The semantics of a CMFS session are defined in terms of a "logical clock" $C(t)$, the "get point" $G(t)$ (the start of the FIFO data) and the "put point" $P(t)$ (the end of the FIFO data). These are byte indices into the file; $C(t)$ is zero when the session starts ($t=0$). Each session has two parameters: R (its data rate) and $\bar{Y}$ (its "cushion"). Figure 2a depicts the semantics of a read session:

- The logical clock advances at rate R whenever it is less than the get point.
- The logical clock stops whenever it equals the get point.
- The put point is always at least $\bar{Y}$ ahead of the logical clock.

These rules imply that if the client removes one byte of data every $1/R$ seconds, it will never block; in other words, the client is guaranteed a data rate of R bytes/second. However, the flow of data need not be smooth or periodic. CMFS promises to stay ahead of the logical clock by a given positive amount (the cushion $\bar{Y}$), and the client's behavior determines how the clock advances. These semantics allow CMFS to handle variable-rate files and other non-uniform access in a simple way. CMFS is guided by client behavior; no explicit rate-control calls are needed, and CMFS need not know about file internals.

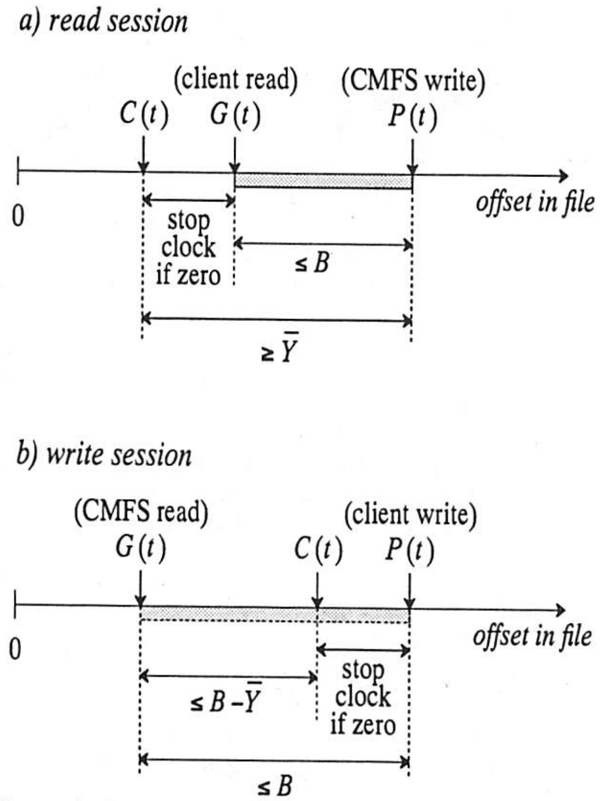


Figure 2: The semantics of read and write sessions are described in terms of a "put point" $P(t)$, a "get point" $G(t)$, and a logical clock $C(t)$. The shaded rectangles represent data in the FIFO.

The semantics of a write session are as follows (see Figure 2b):

- The logical clock advances at rate R whenever it is less than the put point.
- The logical clock stops whenever it equals the put point.
- The get point trails the logical clock by at most $B - \bar{Y}$ bytes, where B is the size of the FIFO.

The Duality of Reading and Writing

In describing the algorithms used by a real-time file system, it is tedious to have to deal with the read and write cases separately; they are similar but they differ in some crucial respects. In CMFS this problem disappears because reading and writing are dual in the following sense. Given a write session, consider a read session for the same file; the FIFO of the read session is identical to the write session, but with "empty" and "full" interchanged (see Figure 3). As shown in [4], the dual read session obeys the rules for read sessions (described above) if and only if the original write session obeys its corresponding rules. (In fact, the semantics of sessions were designed to make this hold.)

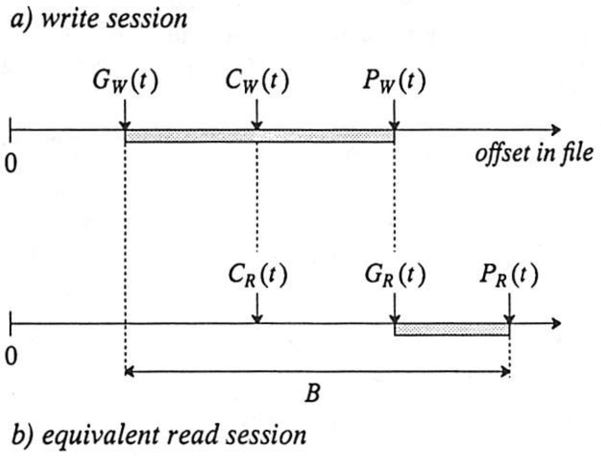


Figure 3: By interchanging empty/full and read/write, a write session (a) is transformed into a read session (b) that is equivalent with respect to scheduling.

Thus, from the point of view of scheduling in CMFS, reading and writing are essentially equivalent. The main difference is the initial condition: an empty buffer for a write session corresponds to a full buffer for a read session. In describing CMFS's algorithms for scheduling and session acceptance, we will refer only to read sessions.

Using CMFS Semantics

The CMFS interface provides an implicit rate control: the client can stop the logical clock by simply not removing data from the FIFO. This control mechanism can serve several purposes:

- CMFS adapts automatically to files that have variable data rate (e.g., variable-rate compressed video) or instantaneous "chunks" such as video frames (this is formalized in [4]).
- A client can pause a session by stopping the removal of data from the FIFO. Performance guarantees will remain valid after the client resumes reading.
- The initial pause that occurs when files from several CMFS servers are played synchronously at a single I/O server (such as ACME [3]) is handled automatically.
- If the hardware is fast enough the client can read arbitrarily far ahead of the logical clock. This "workahead" data can then be buffered (both within CMFS and at other points), protecting against playback glitches and improving the performance of other traffic (see [2]).

CMFS is intended to be part of a real-time distributed system in which each shared resource (CPU, disk, network) can be reserved in "sessions", each of which handles a data stream and has an upper bound on its delay. This "meta-scheduling" scheme is described elsewhere [2]; the connection with CMFS is that a session's "cushion" parameter Y should be at least as large as the delay bound of the resource (usually a CPU) handling the data after removing it from the FIFO. This ensures that the logical clock never stops accidentally due to delay in CPU processing.

The CMFS Control Interface

The operations (RPCs or system calls) to create and start a CMFS session have the following form:

```
ID request_session(
    int direction,
    FILE_ID name,
    int offset,
    FIFO* buffer,
    TIME cushion,
    int rate);

start_clock(ID id);
```

If `direction` is READ, `request_session()` requests a session in which the given file is read sequentially starting from the given offset. If the session cannot be accepted, an error code is returned. Otherwise, a session is established and its ID is returned. `Start_clock()` starts the session's logical clock. The client is notified (via an RPC or exception) when the end of the file has been reached. CMFS also provides a `seek()` operation that flushes data currently in the FIFO and repositions the read or write point.

A real-time file is created using

```
create_realtime_file(
    BOOLEAN expandable,
    int size,
    int max_rate);
```

`expandable` indicates whether the file can be dynamically expanded. If not, `size` gives its (fixed) size. `max_rate` is the maximum data rate (bytes per second) at which the file is to be read or written. CMFS rejects the creation request if it lacks disk space or if `max_rate` is too high.

Non-Real-Time Access

CMFS also supports non-real-time file access. There are two service classes: *interactive* and *background*. Interactive access is optimized for fast response, background for high throughput. The interface for non-real-time access is like that of a UNIX file system, except that the `open()` call specifies the service class. There are no performance guarantees for non-real-time operations.

Disk Layout and Performance Assumptions

To make performance guarantees, we need information about the speed of disk operations, which is determined largely by the disk layout. Many layout policies are possible, and there is no single best policy. For example, contiguous layout minimizes seek time within files, but it may cause external fragmentation, and it is difficult to extend existing files. So instead of adopting a particular policy, we just assume that the disk is read and written in *blocks* of fixed size (a multiple of the hardware sector size), and that the layout has "bounding functions" U_F and V_F:

- For a given file F, $U_F(n)$ is an upper bound on the time to read n logically contiguous blocks of F, independent of the position of the disk head and the starting block number to be read.
- $V_F(i, n)$ is an upper bound on the time needed to read the n blocks of file F starting at block i.

The functions should take into account sector interleaving, interrupt-handling latency, the CPU time used by CMFS itself, and features (such as track buffering) of the disk controller.

Our CMFS prototype uses contiguous allocation. The number of sectors per block is a fixed parameter. Ignoring CPU overhead and other factors, bounds functions for this policy are easy to derive (see [4]). Contiguous layout, however, is only feasible for read-only file systems or if disk space is abundant. For more flexibility, a variant of the 4.2BSD UNIX file system layout [7] could be used. A real-time file might consist of clusters of n contiguous blocks, with every sequence of k clusters constrained to a single cylinder group. n and k are per-file parameters; they are related to the file's `max_rate` parameter.

Admission Control

CMFS accepts a new session only if its data rate, together with the rates of existing sessions, can be guaranteed. One way to decide this is to see whether any static schedule (that cyclically reads fixed numbers of blocks of each session) satisfies the rate requirements of all session and fits in the available buffer space.

To be more precise, suppose that sessions $S_1 \cdots S_n$ read files $F_1 \cdots F_n$ at rates $R_1 \cdots R_n$. A *schedule* ϕ assigns to each S_i a positive integer M_i. CMFS *performs* a schedule by seeking to the next block of file F_i, reading M_i blocks of the file, and doing this for every session S_i. From the functions U and V we can get an upper bound $L(\phi)$ on the elapsed time of performing ϕ.

The data read in ϕ "sustains" S_i for $\frac{M_i A}{R_i}$ seconds, where A is the block size in bytes. $D(\phi)$, the period for which the data read in ϕ sustains all the sessions, is the minimum of these periods. If the data read in ϕ "lasts longer" than the worst-case time it takes to perform ϕ (i.e., if $L(\phi) < D(\phi)$), we call it a *workahead-augmenting schedule* (WAS).

If the amount of data read for each session in a schedule ϕ fits in the corresponding FIFO, we say that ϕ is *feasible*. It is easy to show the following (see [4]): **CMFS can safely accept a set of sessions if there is a feasible workahead-augmenting schedule.**

We now describe an algorithm to compute the *minimal feasible WAS* ϕ (the feasible WAS for which $L(\phi)$ is least). Clearly, a minimal feasible WAS exists if and only if a feasible WAS exists.

Suppose that sessions $S_1 \cdots S_n$ are given. Let D_i be the "duration" of one block of data for S_i, given by A/R_i. Let $\{t_0 < t_1 < \cdots\}$ be the set of numbers of the form kD_i for $k \geq 0$ and $i \geq 0$. Let I_i denote the interval $(t_i, t_{i+1}]$. Let ϕ_i denote the schedule $< \lceil t_i/D_1 \rceil \ \cdots \ \lceil t_i/D_n \rceil >$. Note that ϕ_{i+1} differs from ϕ_i by the addition of 1 block to all sessions whose block durations divide t_{i+1}; hence the sequence of ϕ_i is easy to compute.

The following algorithm computes the minimal WAS (the proof is in [4]):

(1) Let $\phi_0 = <1, \cdots, 1>$ (this is the minimal schedule for which $D(\phi) \in I_0$).

(2) If ϕ_i is infeasible (i.e., there is no allocation $< B_1 \cdots B_n >$ of buffer space to client FIFOs such that $M_i A + Y_i \leq B_i$ for all i) stop; there is no feasible WAS.

(3) If $L(\phi_i) \leq D(\phi_i)$ stop; ϕ_i is the minimal feasible WAS.

(4) Compute ϕ_{i+1} and go to (2).

Disk Scheduling

When a disk block I/O completes, CMFS decides which disk block to read or write next, and it issues the appropriate command (seek, read, or write) to the disk device driver. The algorithm for this decision (the *disk scheduling policy*) must prevent starvation of sessions, and it should handle non-real-time workload efficiently.

As with any disk-based file system, seek overhead is a dominant concern in CMFS scheduling. It is desirable to perform long (multi-block) operations. However, it is only possible to perform long operations if the system is far enough ''ahead of schedule'' so that no sessions will starve before the operation is complete. We use the term *slack time* to mean the maximum time that CMFS can defer doing any reads for sessions.

The slack time, denoted H, is computed as follows. Suppose that the minimal WAS ϕ takes worst-case time $L_1 + \cdots + L_n$, where $L_i = U_{F_i}(M_i)$. Let W_i denote the ''workahead'' for session i; that is, the temporal value of data in the FIFO minus the session's cushion Y. Assume sessions are numbered so that $W_1 \le W_2 \le \cdots W_n$ (this ordering maximizes slack time). If the operations in ϕ is performed immediately in this order, the workahead of session j will not fall below $H_j = W_j - \sum_{i=1}^{j} L(i)$. CMFS can therefore safely defer starting ϕ for a period of $H = \min_{i=1}^{n}(H_i)$.

CMFS factors the disk scheduling policy into three parts:

- A *non-real-time policy* decides whether a non-real-time operation can be initiated.
- If a non-real-time operation is not initiated, a *real-time policy* decides which session to work on.
- A *startup policy* is in effect when sessions have been accepted but not yet started.

We describe these sub-policies separately.

Real-Time Policies

We have implemented and studied several real-time policies (the choice of policy is a CMFS option):

- **The Static/Minimal policy** simply repeats the minimal WAS.
- The **Greedy policy** does the longest possible operation for S_1 (the session with smallest workahead). It reads blocks for S_1 for a period of $H + L_1$; i.e., it uses the entire slack time for workahead on S_1.
- The **Cyclical Plan policy** tries to divide slack time workahead among the sessions in a way that maximizes future slack time. The policy distributes workahead by identifying the ''bottleneck session'' (that for which H_i is smallest) and schedules an extra block for it. This is repeated until H is exhausted. The resulting schedule determines the number for blocks read for S_1; when this read completes, the procedure is repeated.

All policies skip to the next session when a buffer size limit is reached. If at some point all buffers are full, no operation is done; when a client subsequently removes sufficient data from a FIFO, the policy is restarted. In both the Greedy and Cyclical Plan policies, the least-workahead session S_1 is serviced immediately. Therefore the value of H used by these policies can be computed as the minimum of the slack times of all sessions except S_1, yielding **Aggressive** versions of the policies.

Non-Real-Time Policy

Recall that CMFS has two classes of non-real-time traffic: interactive and background. The goal of the non-real-time policy is to provide fast response for interactive traffic and high throughput for background traffic.

A non-real-time operation with worst-case latency L can safely be started if $L \le H$. However, doing so may make H close to zero. This will subsequently force CMFS to do short real-time operations (close to the minimal WAS), causing high seek overhead.

Instead, CMFS uses a *slack time hysteresis* policy for non-real-time workload. An interactive operation is started only if $H \ge H_{I1}$. Once H falls below H_{I1}, no further interactive operations are started until H exceeds H_{I2}. Similarly, background operations are done with hysteresis limits $[H_{B1}, H_{B2}]$. No background operation is started if an interactive operation is eligible to start.

If the hysteresis limits are set appropriately, this policy has two benefits: 1) it keeps slack time from becoming close to zero, and 2) it avoids the seek overhead of rapidly alternating between real-time and non-real-time operations.

Startup Policy

When CMFS accepts a session request, it must delay returning from the `request_session()` call until the system state is ''safe'' (i.e., slack is positive) with respect to the new WAS. CMFS uses the following policy during this startup phase.

Suppose sessions $S_1 \cdots S_n$ are currently active, and session S_{n+1} has been accepted but not yet started. Let ϕ_n and ϕ_{n+1} denote the minimal WASs for the sets $S_1 \cdots S_n$ and $S_1 \cdots S_{n+1}$ respectively. The scheduler enters ''startup mode'' during which its policies are changed as follows:

- Non-real-time operations are postponed.
- For scheduling purposes, slack time H is computed relative to ϕ_n.
- When $S_1 \cdots S_n$ have positive slack with respect to ϕ_{n+1}, a read for S_{n+1} (of the number of blocks given by ϕ_{n+1}) is started. When this read is completed, the state is safe for all $n+1$ sessions. The `request_session()` call for S_{n+1} is allowed to return, ϕ_{n+1} becomes the system's WAS, and the system leaves startup mode. This step is omitted for write sessions because the FIFO of the dual read session (as described earlier) is initially full.

CMFS Performance

The CMFS prototype is written in C++ and runs as a user-level process on SunOS 4.1. Depending on a compile-time flag, disk I/O is either simulated or real. The simulator keeps track of the disk head radial and rotational position, and models disk latencies realistically. Real I/O is done to a SCSI disk via the UNIX raw-disk interface.

Performance on UNIX

We tested a ''real I/O'' version of CMFS on a Sun 4/110 connected to a CDC Wren III via a SCSI interface. To obtain functions *U* and *V*, we measured the time of I/O operations. We found that the time to read a (512 byte) sector on the same track as the previously-read sector, and at a rotational distance of *n* sectors, varied from 6 msecs for $n=11$ to 21 msecs for $n=10$ (i.e., the disk spins 10 sectors before another read command is handled). The average time needed to read a sector at the same rotational position on an adjacent cylinder was 24 msecs, and on an extreme cylinder 54 msecs.

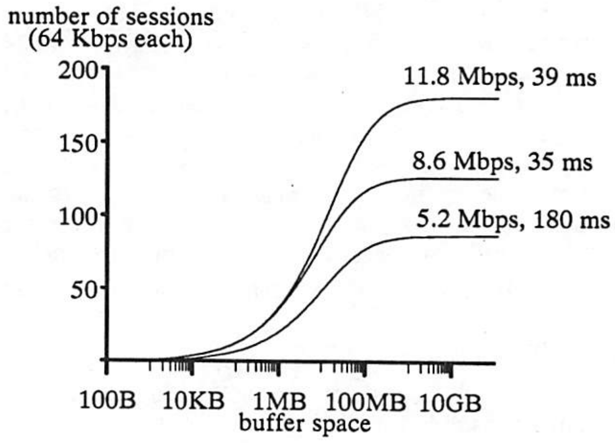
We then measured the CPU time used by CMFS. In a typical situation (1 MB buffer, two 1.4Mbps sessions, background traffic, Greedy policy) the average time per scheduling decision was 250 usecs. Cyclical Plan policy was slightly slower: 303 usecs average. The maximum times (impossible to measure precisely on UNIX) were in the range of 2 to 4 msecs.

Because of the high per-read overhead of UNIX, we used a block size of 35 sectors (1 track). We then defined functions *U* and *V* based on the measured times for UNIX I/O and CMFS execution. With 2 MB of buffer space, this version of CMFS accepts 39 64Kbps (telephone-rate) or 2 1.4Mbps (CD-rate) sessions. Starvation occurs about once per minute, perhaps due to UNIX CPU scheduling. With a workload consisting of 2 CD-rate sessions and random interactive arrivals at a rate of 5/sec., the mean response time is about 80 ms. With 2 CD-rate sessions, the throughput of background traffic is 1.424 Mbps. These results are in rough agreement with the simulator using the same system parameters.

Simulation Studies

Because of the scheduling vagaries of UNIX and the restriction to our available disk drives, the ''real I/O'' version of CMFS is not well-suited to performance studies. Instead, we did simulation-base studies using the CMFS simulation mode. Unless otherwise stated, the simulations use the Cyclical Plan policy and assume a disk with 11.8 Mbps transfer rate and 39 ms worst case seek time. Block size is 512 bytes.

Figure 4 shows the maximum number of concurrent sessions accepted by CMFS as a function of total buffer space, for data rates of 64 Kbps and 1.4 Mbps. Curves are given for three different disk types: 39 ms maximum seek time and 11.8 Mbps transfer rate (CDC Wren V), 35 ms maximum seek time and 8.6 Mbps transfer rate (CDC Wren III) and, 180 ms maximum seek time and 5.6 Mbps transfer rate (Sony 5.25" optical disk).



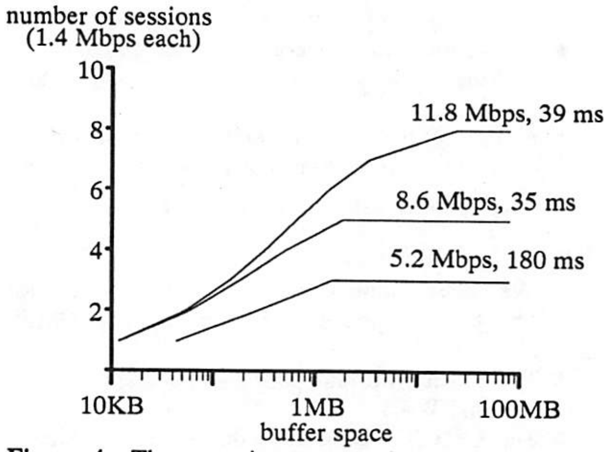


Figure 4: The maximum number of sessions depends on the available buffer space. The disk transfer rate imposes an upper limit on the number of sessions; to reach 90% of the limit with the 11.8 Mbps disk requires 4 MB of buffer space for 1.4 Mbps sessions and 85 MB for 64 Kbps sessions.

An important criterion for real-time scheduling policies is how quickly they increase slack. To study this, we simulated CMFS with three concurrent 1.4 Mbps sessions, no non-real-time traffic, and 8 MB buffer space. From the results (Figure 5) we see that Cyclical Plan performs slightly better than Greedy when slack is low, but that Greedy quickly catches up. Static/Minimal, because it cannot do long operations, performs much worse at higher slack levels. With appropriate hysteresis values CMFS maintains moderate to high slack levels during steady-state operation; thus the dynamic policies are preferable.

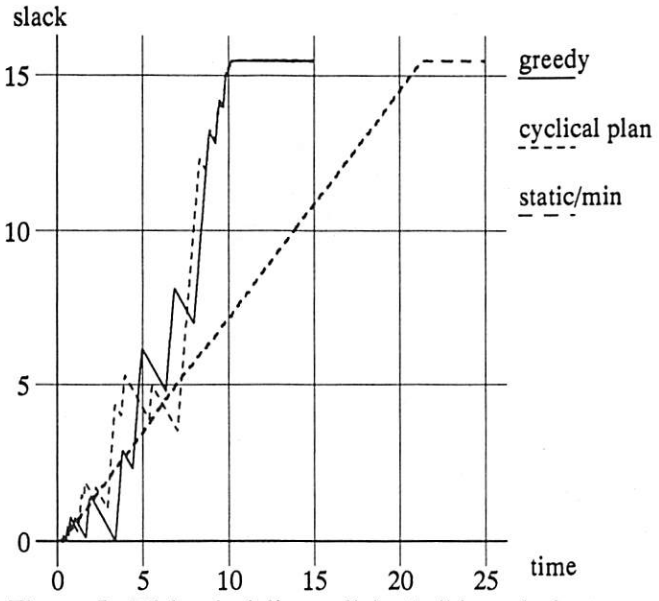


Figure 5: Disk scheduling policies build up slack at different rates. The Aggressive Cyclical Plan (solid line), Aggressive Greedy (dotted) and Static/Minimal (dashed) policies are shown here.

To study the performance of interactive non-real-time traffic, we ran simulations combining Poisson arrivals of interactive requests (each reading a random block) with several sessions. We then measured the average response time of the interactive requests. Details are given in [4]; the main results are:

- The effect of hysteresis parameters is greatest under heavy load; otherwise slack remains high and hysteresis is not exercised.
- Reasonable "rule of thumb" values for the hysteresis parameters are $H_{I1} = .3H_{max}$ and $H_{I2} - H_{I1} = \min(.3H_{max}, 0.5)$, where H_{max} is the upper limit on slack time imposed by buffer space.
- Response time decreases with increasing buffer space; in scenarios involving CD-rate sessions, the "knee" was around 1 MB.

To study the effect of real-time traffic on background traffic throughput, we simulated several sessions and a single background task that sequentially reads a long, contiguously-allocated file. We define the *background throughput fraction* T as the fraction of residual disk throughput (i.e., disk throughput not taken up by real-time sessions) used by the background task. We found that T was maximized for (roughly) $H_{B1} = .25H_{max}$ and $H_{B2} = .9H_{max}$. Again, increasing buffer space improves non-real-time performance; a buffer of at least 1 MB was needed to attain a value of T above 0.9.

Finally, we studied startup time for read sessions. We ran a simulation in which requests for six sessions 1.4 Mbps arrive at time zero. The first session starts in about 0.1 seconds. The gap between the start times of successive session then increases rapidly; the sixth session takes about 2.0 seconds to start. This is because the workaheads of existing sessions have to be increased to accommodate the new minimal WAS, which becomes longer as more sessions are added.

Conclusion

CMFS shows that it is possible for a file system to simultaneously handle multiple sessions with different data rate guarantees, together with non-real-time workload. This is an important step in the integration of audio/video in general-purpose computer systems. CMFS contributes new ideas in its acceptance test and scheduling policies, and also in the flexible but rigorous semantics of sessions.

Our performance experiments show that 1) for real-time traffic, dynamic scheduling policies (such as Greedy and Cyclical Plan) perform best; 2) significant buffer space is needed for the system to perform near the limits of the disk drive; 3) slack-time hysteresis limits can have a large effect on non-real-time performance.

There are many directions for further work in CM file systems: integration of low-level servers like CMFS with higher levels, extension to parallel I/O architectures, dynamic layout and compaction schemes, and improved scheduling policies for non-real-time workload, to name a few.

Acknowledgements

This research was supported by NSF, Sun Microsystems, Sony Corp., and ICSI. Thanks to George Homsy, Vassilios Polimenis and Mark Moran for helpful discussions about CMFS, and to Eric Allman for encouragement and advice.

References

[1] C. Abbott, "Efficient Editing of Digital Sound on Disk", J. Audio Eng. Soc. 32, 6 (June 1984), 394.

[2] D. P. Anderson, "Meta-Scheduling for Distributed Continuous Media", UC Berkeley, EECS Dept., Technical Report No. UCB/CSD 90/599, Oct. 1990.

[3] D. P. Anderson and G. Homsy, "A Continuous Media I/O Server and its Synchronization Mechanism", IEEE Computer, Oct. 1991, 51-57.

[4] D. P. Anderson, Y. Osawa and R. Govindan, "Real-Time Disk Storage and Retrieval of Digital Audio and Video", ACM Trans. Computer Systems, to appear. Also UC Berkeley

EECS Dept. Technical Report No. UCB/CSD 91/646.

[5] J. Gemmell and S. Christodoulakis, ''Principles of Delay Sensitive Multi-media Data Storage and Retrieval'', ACM TOIS, to appear.

[6] R. Govindan and D. P. Anderson, ''Scheduling and IPC Mechanisms for Continuous Media'', Proc. of the 13th ACM Symp. on Operating System Prin., Pacific Grove, California, Oct. 14-16, 1991, 68-80.

[7] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, ''A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3 (Aug. 1984), 181-197.

[8] P. V. Rangan and H. M. Vin, ''Designing File Systems For Digital Audio and Video'', Proc. of the 13th ACM Symp. on Operating System Prin., Pacific Grove, California, Oct. 1991, 81-94.

[9] D. Steinberg and T. Learmont, ''The Multimedia File System'', Proc. 1989 International Computer Music Conference, Columbus, Ohio, Nov. 2-3, 1989, 307-311.

[10] D. B. Terry and D. C. Swinehart, ''Managing Stored Voice in the Etherphone System'', ACM Trans. Computer Systems 6, 1 (Feb. 1988), 3-27.

[11] C. Yu, W. Sun, D. Bitton, R. Bruno and J. Tullis, ''Efficient Placement of Audio Data on Optical Disks for Real-Time Applications'', Comm. of the ACM 32, 7 (1989), 862-871.

Author Information

David P. Anderson has been an Assistant Professor at UC Berkeley since 1985. He is currently studying Macintosh programming and jazz piano. Contact him at anderson@icsi.berkeley.edu or 1627 Blake St., Berkeley CA 94703.

Yoshi Osawa recently spent a year as Visiting Industrial Fellow at UC Berkeley. After graduating from the University of Tokyo in 1985 he has worked for Sony, where his address is Sony Corporation, Electric Devices Group, Storage Systems Business Unit, Magneto-Optical Disk Drive Division, 2255 Okata, Atsugi, Kanagawa 243, Japan. His email address is osawa@strg.sony.co.jp.

Ramesh Govindan received his undergraduate degree from IIT-Madras and is completing his Ph.D. degree in CS at UC Berkeley. He can be reached at Evans Hall, UCB, Berkeley CA 94720 or ramesh@icsi.berkeley.edu.

Mainframe Services from Gigabit-Networked Workstations

J-P. Baud, C. Boissat, F. Cane, F. Hemmer, E. Jagel, A. Kumar, G. Lee, B. Panzer-Steindel L. Robertson, B. Segal, A. Trannoy, I. Zacharov – CERN – Computing & Networking Division

ABSTRACT

Until recently, large mainframes and super-computers were considered essential for powerful scientific batch computing services requiring intensive tape usage, large well-managed disk storage systems, high throughput and maximum reliability. However, this situation has changed dramatically over recent years with the appearance of RISC-based workstations with performance characteristics, at least for scalar computations, comparable with the fastest mainframes but with an order of magnitude better price/performance. At the same time, competitively priced workstation-class disk and tape systems with adequate performance and reliability have become available. Combined with newly-developed LANs and Gigabit networking solutions, it is now possible to provide scalable and integrated *mainframe-class* services on workstation platforms with the UNIX operating system.

Previous papers have summarized CERN's work over the past two years in developing and introducing such services on a large scale. The latest system is called *SHIFT*, or *Scalable Heterogeneous Integrated FaciliTy*. The SHIFT facility performs a wide range of scientific data processing tasks including many with high I/O requirements and is comparable in CPU capacity to the CERN computer center. Similar systems are now being built within the budgets of smaller institutes which previously had to depend on remote university or national computing centers.

The present paper gives a short review of the SHIFT project's goals and architectural principles, and a detailed account of the networking and software design and implementation problems that were encountered and solved.

Background

The work described in this paper was initially motivated by the appearance on the market of inexpensive processors and storage systems, using technology developed for personal workstations, but which had performance characteristics comparable with those of traditional mainframes.

CERN Central Computing Environment

CERN is the European Laboratory for Particle Physics and is host to many physics collaborations using the laboratory's accelerator facilities. Physics data from the experimental particle detectors are recorded by on-line data acquisition systems and written to IBM 3480 cartridges. The data are organized in *events* where the size of an event varies from 10 KBytes to 200 KBytes. Analysis of the raw data is carried out both at the CERN computer center and at collaborating institutes throughout Europe.

At CERN, computer systems used for data analysis are benchmarked in units called *CERN CPU Units* using a representative suite of High Energy Physics codes, written in FORTRAN. For comparison, a VAX 11/780 is rated at about 0.25 CERN Unit. Note that only scalar CPU power is compared in this paper as CERN's workload is not generally vectorizable.

Currently the CERN computer center provides three mainframe services, as shown in Table 1.

Approximately 80% of the mainframe CPU goes to batch work. Most batch jobs require access to tapes. The CERN tape vault houses 150,000 tapes and cartridges with an equal number of tapes stored

Mainframe	CPU (CU)	Disk (GB)	3480 Tapes	8mm Tapes
Cray X/MP-48	32	50	6 manual 4 robotic	
IBM 9000/900	120	400	38 manual 10 robotic	8 manual
Vax 9000-410	9	50	8 manual	

Table 1: CERN – Central Mainframes

outside the vault but in active use. A robot with a capacity for 18,000 3480 cartridges handles approximately 20% of the mount requests. Round-the-clock manual mounts are the responsibility of operations staff.

Into this environment, a batch project based on RISC workstations was initiated two years ago. Beginning with a single APOLLO DN10040, the project has grown substantially and now forms an operational service which exceeds the total deliverable CPU capacity of the central mainframes. The service is collectively known as the *Centrally Operated RISC Environment* or *CORE*, and has three components: *SHIFT, CSF,* and *HOPE.*

SHIFT The SHIFT system forms the subject of the present paper. It is a general purpose facility for jobs with a broad range of I/O requirements and which require access to many Gigabytes of on-line data. SHIFT workstations are networked via both Ethernet and UltraNet. The SHIFT CPU and disk servers are currently SGI Power Series 340 workstations and the tape servers are SUN 4/330s.

CSF The *Central Simulation Facility* or *CSF* is a platform for CPU-intensive work with low I/O requirements. The service runs on 16 HP9000/720 machines which are networked via Ethernet and which have full access to the SHIFT tape service. To the end user, CSF systems are seen as a single batch facility.

HOPE The HOPE service is an earlier system based on 3 APOLLO DN10040 machines. It is for CPU-intensive, low I/O work and it will be phased out during the course of 1992 as HOPE workload is taken over by *CSF.* HOPE is a joint project between *Hewlett-*

Service	CPU (CU)	Disk (GB)	3480 Tapes	8mm Tapes
SHIFT	100	150	6 manual	2 manual 2 robotic
HOPE	50	10		
CSF	150	10		

Table 2: CERN – Central RISC Services

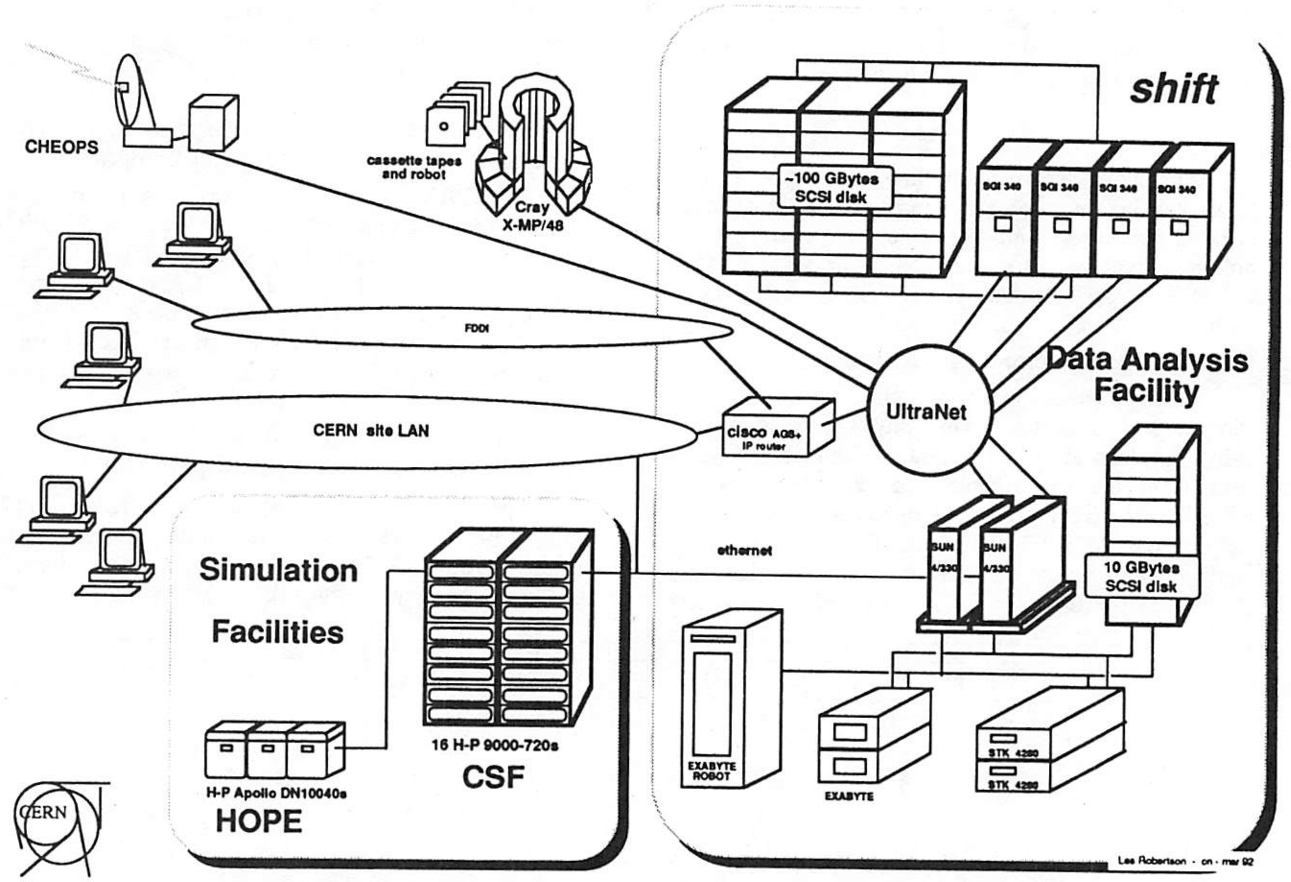


Figure 1: CERN – Centrally Operated RISC Environment

Packard and *OPAL*, a large physics collaboration based at CERN.

The current configuration for the centrally operated RISC-based workstation batch services is given in Table 2, and also indicated in Figure 1.

Project Goals

The goal was to develop an architecture which could be used for general purpose scientific computing, could be implemented to provide systems with excellent price/performance when compared with mainframe solutions, and could be scaled up to provide very large integrated facilities, or down to provide a system suitable for small university departments. The resulting systems should present a *familiar and unified system image* to their users, including access to many Gigabytes of disk data and to Terabytes of tape data: this is what we imply by the word *integrated.*

The goals of the SHIFT development were as follows.

- Provide an INTEGRATED system of CPU, disk and tape servers capable of supporting a large-scale general-purpose batch service
- Construct the system from heterogeneous components conforming to OPEN standards to retain flexibility towards new technology and products
- The system must be SCALABLE, both to small sizes for individual collaborations/small institutes, and upwards to at least twice the current size of the CERN computer center
- The batch service quality should be at least as good as mainframe batch quality, operate in a distributed environment, and have a unified priority scheduling scheme
- Provide automatic control of disk file space, integrated with a tape staging service
- Provide support for IBM 3480-compatible cartridge tapes, Exabyte 8mm tapes, and other developing tape technologies, with access to CERN's automatic cartridge-mounting robots
- System operation and accounting to be integrated into the CERN central computer services
- The architecture should also be capable of supporting interactive scientific applications

SHIFT Architecture and Development

The SHIFT system has been outlined in earlier papers [1,2,3]. A prime goal of the SHIFT project was to build facilities which could scale in capacity from relatively small systems up to several times that of the combined power of the CERN central mainframes. To achieve this, an architecture was chosen which encouraged separation of functionality. This allowed modular extensibility, flexibility, and optimization of each component for its specific function. Figure 2 shows this schematic architecture.

The principal elements of SHIFT are logically divided into CPU servers, disk servers and tape servers, with distributed software which is

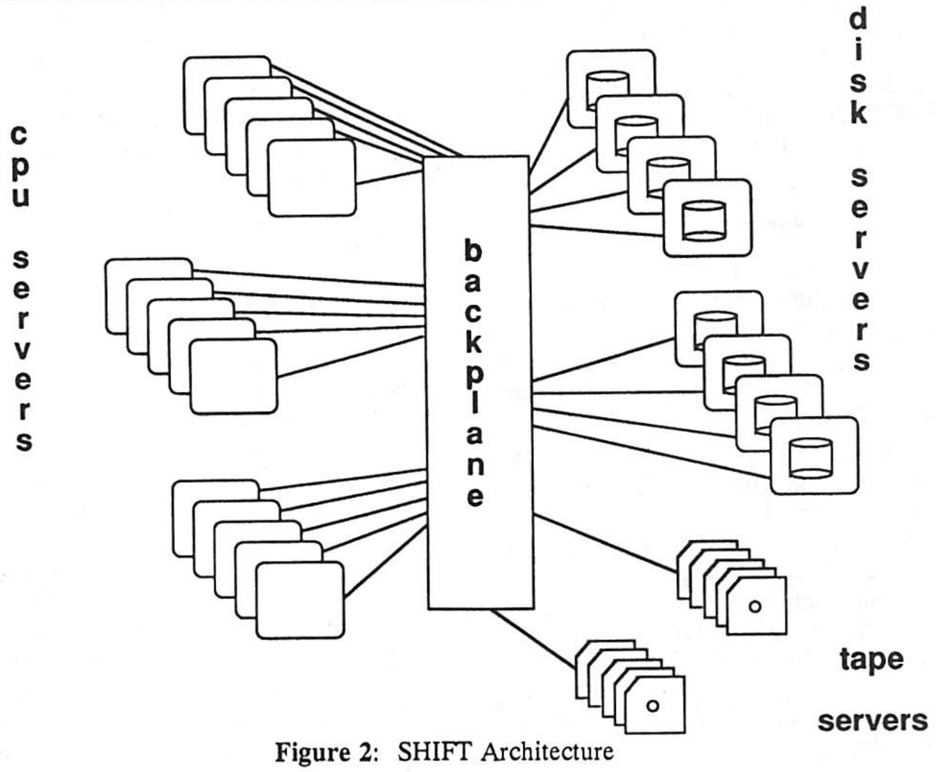


Figure 2: SHIFT Architecture

responsible for managing disk space, staging data between tape and disk, locating staged files, batch scheduling and accounting. These servers are interconnected by the *backplane*, a very fast network medium used for optimized special purpose data transfer. A detailed discussion of the backplane's requirements and properties is given later in the paper. The backplane is connected to the site's general purpose network infrastructure by means of an IP router, providing access to workstations distributed throughout CERN and at remote institutes.

An emphasis throughout the project has been on software portability, to allow flexible choices to be made for hardware platforms for each system component. Such choices can then be made using the most up to date evaluations of currently available products. Addition of further system types to the existing configuration is regularly reviewed. No major difficulties are foreseen in incorporating any UNIX based systems to SHIFT. As an example, a change from DEC to Sun workstations was made very quickly during the development of the system tape servers.

We believe that the modular approach we adopted was also the key to the very short development timescale we achieved. The design studies for the SHIFT project began in mid-1990. In parallel, the technical evaluations of various workstation and networking products were undertaken. By September 1990, code development had begun and orders for hardware had been sent out. The first local tests with SGI Power Series workstations connected via UltraNet took place at the end of December 1990. A full production environment was in place by March 1991. Software and performance improvements were made throughout 1991 and a decision to double the SHIFT CPU, disk and tape capacity was taken in November 1991 and carried out in early 1992.

The Backplane

A critical issue in the SHIFT design turns out to be the need for a high performance network: the *backplane*. Its aim is to provide to CPU servers remote disk and tape I/O facilities with as good performance and as low an overhead as I/O to locally-connected disks and tapes.

Our simulations of SHIFT configurations and workloads were used to compare various modern LAN technologies as backplane candidates. They showed not only that Ethernet was entirely inadequate, but that even FDDI would prevent scaling up to large SHIFT configurations due to its limited total bandwidth, as well as by its low delivered per-interface bandwidths which result from (today's) high CPU and system overheads when running TCP/IP over FDDI. Thus only small SHIFT systems can use an FDDI backplane.

Approximate backplane requirements can be illustrated by the following simple calculation: a medium I/O bound physics analysis job, running on a nominal 1 CERN Unit power CPU, is estimated to read about 20 KBytes/sec of data from disk, and to write up to half of this amount back to disk before completing. This translates to an aggregate 3 MBytes/sec backplane rate for a 100 CERN Unit system doing remote disk I/O. In a worst case scenario, all of this disk data must be staged from tape beforehand, and the results written back to tape afterwards, doubling the backplane aggregate to 6 MBytes/sec. If now heavily I/O bound jobs are included, this will multiply the aggregate still further: to avoid network congestion a backplane peak capacity of 15 MBytes/sec is considered necessary to support a general mix of I/O intensive jobs running on this medium-size 100 CERN Unit configuration. This already exceeds FDDI's possibilities.

Equally significant is the CPU consumption incurred by such data rates using presently implemented FDDI interfaces. Taking the nominal aggregate of 6 MBytes/sec from above, and noting that *two network interfaces participate in each backplane transfer*, a total of 12 MBytes/sec of interface activity is present under medium conditions on a 100 CERN Unit system. Measurements of various FDDI implementations at CERN have shown that between 2 and 6 CERN Units of CPU are needed to drive 1 MByte/sec through todays' FDDI interfaces, translating into a figure between 24 and 72 CERN Units for 12 MBytes/sec, or an average of **50% of installed CPU capacity**.

Finally, the *peak per-interface* data rate available will affect the number of networked units required to assemble a SHIFT configuration. Using the figure of 15 MBytes/sec peak backplane traffic (i.e., 30 MBytes/sec of peak interface traffic) on a 100 CERN Unit system with full tape↔disk staging, we find that the total interface traffic breaks down into 7.5 MBytes/sec each of CPU server and tape server traffic, plus 15 MBytes/sec of disk server traffic. In order to satisfy these rates with a reasonably small number of server modules (say about 3 of each type), we require sustained interface rates of between 3 and 5 MBytes/sec. Today FDDI can achieve only about half of these sustained data rates under normal production conditions, thus forcing the use of many server modules (particularly disk servers).

The current solution for the large SHIFT configuration at CERN is to use UltraNet [4] equipment for the backplane, as this product includes special purpose protocol-processing hardware on each interface, plus several times the FDDI total bandwidth. Equally importantly, it supports the most widespread standard TCP/IP application interface (BSD sockets).

However, the optimal use of UltraNet requires some special understanding. First of all, UltraNet is designed to assist *stream-type* applications and is not very effective for datagram or small packet-size transfers. Thus file access via stream sockets, with large record-lengths, is well supported whereas access via NFS is not. This was well understood from the start, and fitted our model of remote disk and tape I/O on condition that such accesses are sequential and use large record length; this is the case for High Energy Physics analysis programs, which typically use record lengths of 32 KBytes.

Table 3 summarizes the performance characteristics of FDDI and UltraNet, for simple memory↔memory transfers. It shows UltraNet's dependence on blocksize and (in the final column) the relative FDDI and UltraNet CPU costs of data transfer, expressed as the number of MBytes/sec achievable per single fully-loaded CPU. It can be seen that an UltraNet blocksize of 128 KBytes, even under such test conditions, is much more effective than one of 32 Kbytes. During our detailed performance analysis, 128 KBytes was found to be an optimal choice under actual operational conditions.

Software Architecture

Four areas of software development were identified in order that the SHIFT systems could offer a scientific computing environment comparable to that of a conventional mainframe.

Distributed Batch Job Scheduling The *Network Queuing System* (NQS) is used on SHIFT for batch job submission, control and job status enquiry. As the physics workload is located on several machines, the batch jobs must be scheduled evenly across all CPU servers to maximize job throughput and CPU utilization. In addition, users expect to see consistent job turnaround times. To achieve these goals, a load balancing scheme was incorporated into NQS.

Distribution of Files The SHIFT filebase is composed of many distinct UNIX filesystems located on different hosts across the network. Current technologies in distributed file systems are unable to satisfy the demand for data throughput. Moreover, these distributed file systems usually do not allow file systems to spread over more than one machine. A *Disk Pool Manager* (DPM) was developed to manage the SHIFT files and filesystems across the network.

Remote File Access Current distributed file systems have performance limitations when used for demanding applications over high speed networks such as UltraNet. A specialized Remote File Input Output (RFIO) subsystem was developed taking into account the underlying network characteristics.

Sink	Source	Blsize(KB)	MB/sec	CPU (Sink)	MB/1CPU (Sink)
UltraNet:					
SGI 340S	SGI 320S	10	2.8	35%	8
		32	4.7	21%	23
		128	8.8	14%	62
		256	10.5	12%	84
		512	11.3	9%	122
		1000	11.8	8%	140
Cray/LSC	SGI 340S	20	3.1	4%	80
		200	5.5	1%	550
		2000	6.0	1%	600
Sun4/330	SGI 340S	20	2.5	20%	13
		200	3.4	14%	24
		2000	3.5	13%	27
FDDI:					
SGI 320S	DEC 5200	32	2.2	40%	5.5
SGI 320S	Sun4/670	32	3.0	68%	4.5
Sun4/670	SGI 320S	32	3.2	60%	5.3
Sun4/670	DEC 5200	32	2.4	45%	5.2
DEC 5200	SGI 320S	32	2.1	80%	2.6
DEC 5200	Sun4/670	32	1.7	65%	2.6

Table 3: UltraNet vs. FDDI Performance (memory↔memory)

Tape Access High Energy Physics computing at CERN makes extensive use of IBM 3480 tape cartridges for the storage of data from experiments. A 3480 cartridge has a capacity of 200 MBytes. Exabyte cartridges are also used to a smaller extent. These have up to 5 Gigabyte capacity but with a data transfer rate substantially lower than the 3480.
UNIX systems have been traditionally weak in the area of tape support, one exception being the Cray UNICOS system. A portable tape subsystem was developed to manage a range of tape devices attached to different hosts. In addition, a remote tape copy utility, RTCOPY was developed which could be used in the distributed environment.

The subsequent sections describe each of the four areas in more detail.

Batch System Enhancements

The *Network Queuing System NQS* is a facility for job submission and scheduling across a network of UNIX batch workers. At CERN, it has been ported to numerous workstation platforms and useful enhancements have been added such as limits on the number of jobs run for any user at one time, an interactive global run limit, the ability to move requests from one queue to another and the ability to hold and release requests dynamically. Moreover, CERN has implemented in NQS the ability to have the destination server chosen automatically, based on relative work loads across the set of destination machines. Users submit jobs to a central pipe queue which in turn chooses a destination batch queue or *initiator* on the least loaded machine that meets the jobs' resource requirements. If all initiators are busy, jobs are held in the central pipe queue and only released when one becomes free. In addition, a script running above NQS holds or releases waiting jobs with a priority based on their owner's past and current usage of the SHIFT service.

Disk Pool Manager

The SHIFT data filebase comprises many UNIX filesystems which are located on any of the SHIFT hosts across the network. In order that users see a unified data file space, the notion of a *pool* was created. A *pool* is a group of one or several UNIX filesystems and it is at the *pool* level that file allocation is made by the user. Pools can be much larger than conventional UNIX filesystems even where logical volumes are available. Pools may also be assigned attributes. For example, a pool used for staging space can be subject to a defined garbage collection algorithm. The pools in SHIFT are all managed by the *Disk Pool Manager.* The *Pool Manager* balances disk space when creating new files and directories and it may be used to locate and delete existing files.

The interface to the *Disk Pool Manager* is via UNIX user commands. The **sfget** command allocates a file of a given size within a specified pool. The command returns a full path name for the file based on the convention that all SHIFT file systems are mounted globally with NFS on the mount point */shift/<host name>*. If the file requested already exists within the pool, **sfget** simply returns the path name without allocating any space. Other commands are provided to list, remove and manage files. In addition, a user-callable garbage collector has been implemented which maintains defined levels of free space in a pool. This is useful for physics data staging where data are copied from tape to disk before being accessed by user programs.

The design of a central *Pool Manager* has proved limited and, in particular, did not scale well with the rapid growth of the filebase. The first implementation was for a filebase of 40 Gigabytes whereas there are currently over 100 Gigabytes connected. In addition, problems arose from the fact that not all users were using **sfget** to allocate files and file system usage grew outside the control of the Pool Manager. To counter this problem, file system scans were incorporated to reflect the actual status and the results were stored in a centrally managed table. Overall performance is still a problem with the centralized *Disk Pool Manager* and a project is now under way to rewrite the software using a more distributed approach.

Remote File I/O System

The Remote File I/O system (RFIO) provides an efficient way of accessing remote files on SHIFT. Remote file access is also possible using NFS but RFIO takes account of the network characteristics and the mode of use of the files to minimize overheads and maximize throughput. RFIO maintains portability by using only the BSD socket interface to TCP, and thus operates over UltraNet, Ethernet, FDDI or other media. RFIO transmits I/O calls from client processes to remote RFIO daemons running on all SHIFT hosts.

RFIO is implemented with both C and FORTRAN interfaces. In C, the system presents the same interface as local UNIX I/O calls: **rfio_open** opens a file like **open(2)**, **rfio_read** reads data from a file like **read(2)** etc. Most High Energy Physics programs are written in FORTRAN, and usually interface their I/O via one or two intermediate library packages. RFIO has been incorporated into these, so its usage becomes completely transparent to the users of these programs.

RFIO was treated as one of the key performance factors of SHIFT. When a detailed investigation of system performance was undertaken, a major effort was made to reduce the operating system overheads incurred by RFIO. This is described in the section below on System Performance.

Magnetic Tape Support

High Energy Physics computing makes extensive use of IBM 3480 and Exabyte 8mm tape cartridges. The initial approach for tape access on SHIFT was to access tape units connected to the Cray UNICOS system. Subsequently, a portable UNIX Tape Subsystem was designed to satisfy all SHIFT's requirements in the area of cartridge tape access. The subsystem runs on all SHIFT hosts to which tape devices are connected.

Portable UNIX Tape Subsystem

UNIX systems usually offer a primitive tape interface which is not well adapted to a multiuser environment. Four basic functions are typically provided:

- **open**(2)
- **read**(2)
- **write**(2)
- **close**(2)

Several **ioctl**(2) commands are also provided but there is no operator interface, label processing, or any interface to a tape management system. The SHIFT Tape Subsystem offers dynamic configuration of tape units, reservation and allocation of the units, automatic label checking, an operator interface, a status display and an interface to the CERN/Rutherford Tape Management System. It is written entirely as user code and does not require any modification of manufacturers' driver code. It currently supports StorageTek's 4280 SCSI tape drive (an IBM 3480 compatible) as well as Exabyte 8200/8500 drives.

Automatic tape file labelling is not provided as this can only be done by modifying the tape I/O driver. Instead, a set of user callable routines were written to perform the same task. In practice, most tape I/O is done by using a tape staging utility, RTCOPY which hides details of these routines from the user.

Tape Copy Utility, RTCOPY

To provide tape access for every SHIFT CPU and disk server, a tape copy utility RTCOPY was developed which allows tape access across the network. Internally RTCOPY uses RFIO software to maximize the data transfer speed and thus minimize the tape unit allocation time. RTCOPY intelligently selects an appropriate tape server, by polling all known tape servers to query the status of their tape unit(s). RTCOPY supplies any missing tape identification parameters by querying the Tape Management System as needed. RTCOPY then initiates the tape copy, informs the user when the operation is complete, and deals with error recovery.

Software Maintenance and Distribution

The SHIFT software is distributed to many sites outside CERN and the distribution and maintenance across different platforms presents a new challange. We currently support SGI Irix 3.3.3, Sun3/SunOS 4.1.1, Sun4/SunOS 4.1.1, Sun 4/SunOS 4.1.2, UniCOS 6.1, DomainOS, VAX/Ultrix 3.x, DecStation/Ultrix 4.x, RS 6000/AIX 3.2, HP 9000/HP-UX 8.05.

Our experience has shown that UNIX tools like **make**(1) and **sccs**(1) are only a partial solution to the problem of software maintenance in a heterogeneous environment. For example, **make** does not support conditional rules, and subtle differences in operating system versions are hard to deal with. We are currently investigating the **imake**(1) tool used by the X11 consortium for maintaining our software suite.

System Performance

During the design phase of the project, performance estimates were made using a straight-forward simulation program using data obtained from benchmark programs running on limited test configurations (made available by potential suppliers and other organizations). When our own hardware was installed, these tests were repeated, placing a great deal of emphasis on what we believed would be the major performance issue, the UltraNet performance. Our initial configuration had only two disk channels and we were therefore unable to perform full-scale disk performance tests and were content to extrapolate the disk performance from simple tests.

These tests led us to assume that we would be able to support an aggregate (multi-stream) data rate between a two processor SGI 4D/320S disk server and a four processor SGI 4D/340S CPU server in excess of 6 MBytes/second with less than 60% utilisation of either the disk server's network interface or its CPUs (and therefore incurring no serious queueing problems). In practice we installed a four processor disk server, and so assumed that this aggregate performance target would be easily achievable; we thought that our second performance target, a single stream remote disk data rate approaching that of a local disk, would be much more difficult.

When SHIFT was initially commissioned, the job mix submitted was more or less as had been expected and the performance was adequate, satisfying the aggregate demand of about 2 MBytes/second. However, due to a change in physics emphasis after some months, many more I/O intensive jobs began to arrive and, in spite of the presence of the UltraNet backplane, the networking performance was now found to be quite disappointing. Instead of rising according to our estimates, we were seeing network data rates saturating at just over 2 MBytes/sec on the running systems (which were of course heavily loaded with batch computation and disk and Ethernet I/O), and at about 4 MBytes/sec when doing multi-stream RFIO under test conditions between unloaded SGI 4D/340's.

Investigation showed that the UNIX system load had become the factor limiting performance. This was much higher than we had predicted, and we assumed that it was due to the RFIO protocol and the architecture of the RFIO server. We therefore began a series of improvements in these areas.

The system overhead due to UltraNet is to a good approximation the same when transferring a few bytes or hundreds of kilobytes. We also noticed that there is a strong dependence of UltraNet performance on network block size, and we showed that in a general operating environment there was optimal performance when using a 128 KByte block. This led to the implementation of a buffered mode of operation for RFIO reads, which now transfer 128 KBytes across the network when using UltraNet.

At the same time we embarked on major refinements to the RFIO protocol, with the aim of minimizing the number of system calls required. The initial version of the RFIO protocol had been very straightforward and robust, simply mapping local FORTRAN and C file I/O calls to remote calls on RFIO daemons. This resulted in many small-size network transactions (e.g those mapping file seeks, I/O completions, etc.) being interspersed with transfers of user data.

The computing workload which we expect to support has a number of special features:

- the application record size is normally 32 KBytes;
- more than 90% of I/O operations are reads;
- most programs either read records from a file sequentially, or in a *skip-sequential* mode. In the latter case, the program uses a directory which contains some physics characteristics of each of the records in the file and pre-selects a list of the records which are of interest. The program then processes this list sequentially.

The RFIO protocol was thus re-designed with the following improvements:

- As far as possible control messages were eliminated by piggybacking them along with the data transfers.
- Three read access modes which could be specified by the user were defined: *sequential, pseudo-sequential* and *random.*
- Sequential file read access was optimized by using buffered read-ahead with 128 KByte data blocks: the server simply loops, reading from the disk and writing 128 KBytes to the network.
- Pseudo-sequential reading of files uses a *pre-seek* procedure call which enables the user to provide a vector containing a list of {record address, record length} pairs. This list is forwarded to the server, which pre-reads and blocks up the required data. The RFIO client returns it to the user program in response to appropriate seek and read requests.
- In random mode, the client requests each record in turn from the server as directed by the application program.
- The mode selected by the user is merely *advisory* in the sense that the result is functionally correct even if the application changes mode without informing RFIO. The mode selection is required only for performance.

Tests showed that a factor of about two had been gained in RFIO performance as a result of this work. Most of this improvement was due to the use of record buffering and large 128 KByte blocks (which of course itself implies a certain level of *read-ahead*).

Table 4 shows the aggregate data rates (Mbytes/sec) achieved for two modes of transfer for the initial and improved versions of RFIO:

Under test conditions, we were now able to achieve our target data rate (aggregate 6-7 MBytes/second), but this required so much of the disk server CPU that we could not achieve it under realistic production conditions. The RFIO protocol was now as simple as the test programs used in our initial benchmarks, and so we realized that our original method of estimating the performance must be

Number of streams	1	2	4	8
RFIO Version1 (32 KByte buffers)				
sequential	0.8	1.1	1.4	1.5
random	0.6	0.9	1.2	1.3
RFIO Version2 (128 KByte buffers)				
sequential	1.1	2.3	4.4	5.7
random	0.9	1.9	2.6	3.7
RFIO Version3 (128 KByte buffers + fewer system calls)				
sequential	1.2	2.4	5.1	6.7
random	1.2	2.2	3.5	6.0

Table 4: Aggregate RFIO Data Rates (Mbytes/sec)

flawed. We had been blinded by the assumption that the difficult task was the network performance, and we had neglected to study the disk performance. Only at this stage did we carry out full scale disk performance tests, and this immediately gave us the answer to the problem. On the main SHIFT disk servers (SGI multi-processor Power Series 4D/340 systems running IRIX 3.3.3) it was found that the CPU cost to perform a given unit of disk I/O (e.g., reading 1 MByte) increases with the system load. After detailed studies by the manufacturer, this is believed to be due to contention for the internal bus linking CPUs and memory. The load on the bus can be reduced by using *direct* I/O. This method circumvents normal file system operations and avoids the copy from kernel buffer to application buffer. Table 5 shows the improvement in both aggregate data rate and, more significantly, in CPU cost which is now constant.

Table 6 shows the results of a series of tests which read from disk and write their data to UltraNet using filesystem I/O and direct I/O. We are investigating how to exploit direct I/O within RFIO in a production environment.

Current Service at CERN

The SHIFT service at CERN currently has about 400 registered users from two large physics collaborations. Scripts have been implemented to handle most of the repetitive tasks such as account creation, automatic code updates, architecture-independent compilation and linking, tape staging, remote job submission, job query and file transfer, system and user accounting and so on. Usage of the service is expanding. Table 7 summarizes some of the current service characteristics:

As the SHIFT configuration expands, the number of physics groups given access will also grow. It is expected that the SHIFT capacity will again double this year, with extra CPU, disk and tape servers being added in a modular way.

Conclusions

We recognized from the start of the project that networking performance was a major challenge if SHIFT were to be able to handle I/O intensive problems. But we had not realized that *free protocol processing* was not the only answer to that problem; in fact operating system overheads remain the major challenge. Our initially simple remote file access

Number of streams	1	2	4	6	8
Using File System I/O					
Aggregate MBytes/sec	1.8	4.1	7.3	7.8	7.5
CPU cost sec/MByte	.15	.18	.29	.37	.41
Using Direct I/O					
Aggregate MBytes/sec	1.8	3.8	7.1	9.2	10.8
CPU cost sec/MByte	.04	.04	.05	.06	.06

Table 5: CPU Cost of Disk I/O

Number of streams	1	2	4	6	8
Using File System I/O					
Aggregate MBytes/sec	1.8	3.5	4.3	4.7	5.1
CPU cost sec/MByte	.20	.24	.27	.32	.40
Using Direct I/O					
Aggregate MBytes/sec	1.5	2.8	4.9	6.5	7.4
CPU cost sec/MByte	.08	.10	.10	.11	.13

Table 6: CPU Cost of Disk and Network I/O combined#

Users per day	50
Batch jobs per day	150
Tapes staged per day	150
Data staged per day (Gigabytes)	25
MTBI (hours) 1 Apr 91 - 1 Apr 92	150
Percentage CPU Currently Utilized	50

Table 7: Current SHIFT Service Profile

protocol, implemented on high-performance UltraNet sockets, required fundamental modification and tuning, taking advantage of some characteristics of our user applications, before it reached acceptable performance. Moreover, to maximize total aggregate throughput, it is necessary to bypass the traditional UNIX file system handling.

Another problem area we have encountered is that of disk unreliability and repair. A fifty thousand hour MTBF figure sounds good for a SCSI disk unit, but with over a hundred such disks installed this translates to a failure every few weeks. We have learned that we need RAID technology and/or disk mirroring techniques to deal with such issues.

Overall, the system's users consider SHIFT to be successful and are increasing their investments in such equipment. The CERN system is running a wide variety of physics production jobs, and has confirmed our belief that such an approach is entirely practical and economic for many physics computing applications. Even though far from being fully loaded, the current SHIFT is processing about 8,000 CERN CPU Unit-hours of work per week, for which it is mounting over 1000 tapes (and transferring about 150 Gigabytes) of physics data per week. The associated systems CSF and HOPE are processing an additional 20,000 CERN CPU Unit-hours of low I/O work per week. For comparison, the CERN central mainframes deliver a total of about 20,000 CERN CPU Unit-hours per week.

The CERN centrally operated RISC facilities are already delivering one and a half times as much physics computing as the conventional mainframe systems. We consider that the SHIFT goals listed earlier in this paper have been met, and that inexpensive RISC based workstations, suitably deployed, can now be used to provide reliable large scale scientific computing services.

Acknowledgements

Important contributions to this work were made by Alfred Lee, Thierry Mouthuy, and Steve O'Neale of the OPAL physics collaboration, and by Julian Bunn of CERN-CN in simulation studies. We also thank Gail Hanson of Indiana University and David Williams, CERN-CN Division Leader, for their support, and HP-Apollo for their generous contribution of equipment.

References

1. "SHIFT, the Scalable Heterogeneous Integrated Facility for HEP Computing", J-P. Baud et al., Proc. Conference on Computing in High Energy Physics, (CHEP 91), March 1991, Tsukuba, Japan. Universal Academic Press.
2. "Scalable Mainframe Power at Workstation Cost", J-P. Baud et al., Proc. Spring 1991 EurOpen Conference, May 1991, pp. 113-122.
3. SHIFT User Guide and Reference Manual", J-P. Baud et al., CERN-CN, 1211 Geneva 23, Switzerland.
4. UltraNet – An Architecture for Gigabit Networking", R. Beach, Proc. IEEE Conference on Local Computer Networks, September 1990.

Author Information

Jean-Philippe Baud has a background in databases and systems programming for supercomputers. He is responsible for the design and implementation of the portable tape software. Contact him via e-mail at baud@cernvm.cern.ch.

Christian Boissat is a systems programmer in the software group at CERN. He is responsible for development of the Network Queueing System, NQS. His e-mail address is boissat@vxcern.cern.ch.

Fabrizio Cane is a CERN fellow and he implemented the Disk Pool Management software. His e-mail address is cane@vxcern.cern.ch.

Frederic Hemmer is a database and distributed computing specialist who joined CERN in 1984. He is responsible for the overall software architecture, and its continuing development and maintenance. His e-mail address is hemmer@sun1.cern.ch.

Erik Jagel joined CERN as a physicist in 1988. He shared responsibility for SHIFT design and simulation and he now works in the computer center where he is in charge of CSF and HOPE operations. His e-mail address is jagel@cernapo.cern.ch.

Ashok Kumar is a computer scientist specializing in real time systems and parallel computing. He is responsible for performance measurement and analysis and for porting the RFIO subsystem to Ultrix and AIX platforms. His e-mail address is ashok@sun1.cern.ch.

Gordon Lee is a UNIX systems specialist in the User Support Group at CERN. He is responsible for all system administration of the SHIFT configuration. His e-mail address is gordon@cernvax.cern.ch.

Bernd Panzer-Steindel is a physicist working in the OPAL collaboration. He has been actively involved in using and developing the SHIFT service.

Les Robertson has spent 18 years at CERN, in various technical and managerial positions. He is currently the leader of a series of projects, including SHIFT, using RISC computers to provide scientific computing services. His e-mail address is les@cernvm.cern.ch.

Ben Segal is a networking and distributed computing specialist, at CERN since 1971. He was responsible for the SHIFT network development. His e-mail address is ben@cernvax.cern.ch.

Antoine Trannoy spent 15 months as a visitor to CERN. He wrote the tape copy software and enhancements to the RFIO package. He is currently employed at *Centro Ricerche Sviluppi et Studi Superiori* in Sardinia, Italy. His e-mail address is trannoy@cernvm.cern.ch .

Igor Zacharov is a physicist who was a CERN fellow from 1989 to 1991. He now works for Silicon Graphics and is responsible for the CERN account. His e-mail address is zacharov@cernvax. CERN is located at 1211 Geneva 23, Switzerland.

A Highly Available Lock Manager For HA-NFS

Anupam Bhide – IBM T. J. Watson Research Center
Spencer Shepler – IBM Austin

ABSTRACT

This paper presents the design and implementation of a highly available lock manager for highly available NFS (HA-NFS). HA-NFS provides highly available network file service to NFS clients and can be used by any NFS client without modification. This is provided by having two servers share dual-ported disks so that one server can take over the other's disks and file systems if it fails. Making the NFS service highly available is not enough since many applications that use NFS also use other services provided with NFS such as the network lock manager. We describe a scheme whereby each server transfers enough of its lock state to the other so that if it fails, the other server can go through a lock recovery protocol. Our design goal was to make the overhead of transferring the state during failure-free operation as low as possible.

1. Introduction

This paper presents the design and implementation of a highly available lock manager for a Highly Available Network File Server (HA-NFS) [3]. HA-NFS provides tolerance to file server, disk and network failures and can be used by any NFS client. Recovery from server failure is provided by having two servers share access to dual-ported disks and provide backup service for each other. These servers are therefore referred to as twins of each other. However, it is not enough to recover the file server state at a backup server in case of a crash. Most NFS implementations are accompanied by a network lock manager so that clients can obtain locks for files that are remotely mounted. NFS file locking is an extension of local file locking and was designed so that applications can use file locking without having to know whether the file is local or remote. Most NFS implementations support a *lockf()/fcntl()*, System V[1] style of advisory file and record locking over the network. A number of applications use the network lock manager to synchronize access to shared files and to prevent multiple processes from modifying the same file at the same time. Since locking is inherently stateful and NFS is supposed to be stateless, the lock manager is implemented separately from NFS.

When the primary server fails, the lock state must be recovered at the backup server. This paper will describe a design for recovering the locking state at the backup in case of server failure and for enabling a failed server which is recovering and re-integrating to regain its locking state.

We have implemented a prototype of the Highly Available lock manager for HA-NFS on the same platform on which HA-NFS was implemented: a network of workstations and two file servers from the IBM RISC System/6000 family of computing systems running the AIX Version 3 (AIXv3) operating system, and connected by either a 10 Mbit/s Ethernet network or a 4 Mbit/s or a 16 Mbit/s token ring network. We constructed dual-ported disks from off-the-shelf SCSI disks attached to a SCSI bus that is shared by the two servers. The prototype is operational and has satisfied the design goals.

In section 2, we present background information on HA-NFS. In section 3, we describe the NFS locking protocol. In section 4, we describe various design alternatives to enable lock state to survive processor failure and recovery events. Section 5 describes the design we chose and why. Section 6 describes the re-integration protocol executed when a failed server recovers. Section 7 presents an evaluation of the design from the point of view of implementation effort and performance. Section 8 describes the technique used to recover from media and network failures. Section 9 compares our approach with other approaches and section 10 proposes some items for future work.

2. The Highly Available Network File System (HA-NFS)

Traditional approaches for providing reliability in networked file systems use server replication. HA-NFS differs from traditional approaches in that it tolerates server failures by using dual-ported disks that are accessible to two servers, each acting as a backup for the other and hence are called twins of each other. The disks are divided into two sets, each served by one server during normal operation. Each server maintains on *its* disks enough information to reconstruct its current volatile state. Since NFS is an almost stateless protocol, the only volatile information is the duplicate cache information that is needed to detect duplicate transmissions. For example, a "create new file" remote procedure call (RPC) may reach a server and the file create operation may take place, but the acknowledgement to the client could be lost. The client would re-try the RPC and may receive an error because the file already exists as a result of the previous RPC unless the RPC was flagged as a re-try. To detect this, the server stores a cache of recently executed RPCs called the "duplicate cache". For further discussion of this topic see [4].

The two servers periodically exchange liveness-checking messages. If one server fails, the failed server's disks will be taken over by its twin server. The twin then reconstructs the lost volatile duplicate cache state using the information on disk. Then the twin *impersonates* the failed server by taking over its IP address and operation continues with a potential reduction in performance due to the increased load. The clients on the network are oblivious to the failure and continue to access the file system using the same address. During normal operation, the servers communicate only for periodic liveness-checking. The servers do not maintain any information about each other's volatile state or attempt to access each other's disks during normal (failure-free) mode of operation. HA-NFS adheres to the NFS protocol standard and can be used by existing NFS clients without modification.

HA-NFS is implemented on top of the AIXv3 log-based file system. The AIXv3 file system provides serializable and atomic modification of file system meta-data by using transactional locking and logging techniques. File system meta-data are composed of directories, inodes, and indirect blocks. Every AIXv3 system call that modifies the meta-data does so as a transaction, locking meta-data as they are referenced, and recording the changes in a disk log before allowing the meta-data to be written to their "home" locations on disk. In the case of system failure, the meta-data are restored to a consistent state by applying the changes contained in the log. The reliability of ordinary files is ensured by NFS semantics, which require forcing the file data to disk before sending an acknowledgement to the client. The volatile state at an NFS server consists of the duplicate cache; this information is recorded on the disk log so that it can be recovered by the backup server. Further details about the design and the implementation of HA-NFS can be found in [3].

3. The NFS Locking Protocol

In this section, we will provide and overview of the NFS/ONC locking protocol. The locking protocol is implemented outside of the NFS protocol, because the NFS locking protocol is stateful and NFS is designed to be stateless. In most implementations the file locking protocol is actually implemented in two daemons. The daemons are usually named rpc.lockd and rpc.statd and these are the names used in AIXv3. The rpc.lockd daemon at a server handles locking requests for NFS clients which are accessing files at the server. The rpc.lockd daemon acts as a surrogate at the server for client processes and keeps track of what locks are held by clients at any one point in time. At a client, the rpc.lockd keeps track of what locks are held at the NFS server by the various processes.

The second daemon, the rpc.statd daemon at a server keeps a list of clients that are to be tracked for system failure. Similarly at a client, this daemon keeps track of what remote servers are currently being accessed by file lock requests.

Getting A lock

When an application at a client makes a system call requesting a lock on a NFS-mounted file, the client kernel makes a RPC to the client's rpc.lockd. This rpc.lockd then sends the lock request to the rpc.lockd at the server which makes a lock request to the server's kernel. The server's kernel accepts the lock request and returns the appropriate response to the server's rpc.lockd. The server's rpc.lockd will then respond to the client's rpc.lockd with the result. It in turn will respond to the client's kernel which will then return the response to the application.

When the client's rpc.lockd receives the original lock request from the client kernel, it will register the server's host name with the client's rpc.statd. This is done before the lock request is sent to the server's rpc.lockd to be processed. Upon receiving the lock request from the client the rpc.lockd at the server will register the host name of the client with the rpc.statd at the server. The rpc.statds on both the client and server record the host names on disk so that it can be accessed after a failure. This registration process is done for the first lock request only. The rpc.lockd keeps internal state about which hosts it has registered with the rpc.statd so this initial registration step is skipped on subsequent lock requests.

Recovery Actions Upon Client/Server Failure

In a standard NFS (not HA) server configuration, there is a method to rebuild the locking state that is kept by the NFS server. Rebuilding of state occurs only after server failures. It is not needed after client failure and recovery since the applications that took the locks no longer exist after the client's system failure. The only actions that need to be taken when a failed client recovers is to tell relevant servers to release the locks that are held on the client's behalf.

The following explains the corresponding steps that the NFS server rpc.lockd/rpc.statd follow to recover locking state. The rpc.statd is started before the rpc.lockd during system initialization. When the rpc.statd on a server restarts, assuming system failure, it reads from disk the names of systems it was monitoring during its previous incarnation. The rpc.statd then informs each of the rpc.statds on these client systems about the server's failure. Client rpc.statds then inform their rpc.lockds about a server failure.

In the case where the rpc.lockd process on the server has failed, the rpc.lockd process during initialization will inform the local rpc.statd that it had failed and goes into a grace period in which it accepts only lock reclaim requests from clients. When a client rpc.lockd is informed of server failure, it goes through its lock table and re-requests or reclaims all locks it had held at that server. After all clients go through this protocol, the server has now regained the lock state that was held before the failure.

In the case where the rpc.lockd process on the server has failed, the rpc.lockd process during initialization will inform the local rpc.statd that it had failed and goes into a grace period in which it accepts only lock reclaim requests from clients. When a client rpc.lockd is informed of server failure, it goes through its lock table and re-requests or reclaims all locks it had held at that server. After all clients go through this protocol, the server has now regained the lock state that was held before the failure. If the client system fails, the rpc.statd at the client will notify the servers of client failure. The list of servers is built from the list of monitored servers that the rpc.statd was keeping in the file system of the client. Upon notification of client system failure, the server's rpc.lockd will release all of the file locks that were held by that client.

4. Design Alternatives

To design a highly available lock manager for HA-NFS, a way must be found to transfer the locking state held by the primary HA-NFS server to the backup or twin HA-NFS server. This needs to be done so that correctness can be maintained in the operation of the NFS server from the client's perspective.

The first approach that might be taken is to follow the same general scheme for transferring the lock state that the HA-NFS server uses to transfer file system state and duplicate cache entries to the twin HA-NFS server. Recall that duplicate cache entries are used to detect request re-transmissions that occur when acknowledgements get lost. The HA-NFS server stores the duplicate cache entry in the file system log when the duplicate entry is initially added to the duplicate cache table. This works because of the one to one mapping of the duplicate cache entry and the commit of the entry to the AIX Journaled File System (JFS) log. For this same mechanism to work for the NFS locking there needs to be a mapping between the lock/unlock operations of the client and JFS logging commit points which correspond to metadata modification points. This mapping does not exist and would not be possible without a redesign of the logging services to serve other than normal JFS activity. This would also mean that locking operations would run at disk speed.

The second approach could be to have the rpc.lockd of each HA-NFS server transfer the locking state to the twin HA-NFS server. This would need to be done with each positive response to a client's locking request. Before the primary server's rpc.lockd sends its positive/granted response to the client, it would have to call the rpc.lockd on the twin HA-NFS server. The rpc.lockd on the twin could then build the same locking state as the primary server. This approach would have a performance impact on each locking operation. This would also affect the twin's locking performance and general system performance since it would be fielding the same locking requests that the primary would be handling. Another drawback to this approach would be the added complexity of keeping locking state for the twin and differentiating that locking state from the local locking state of the twin.

The third approach is similar to the second, except that instead of all positive lock/unlock responses being passed to the twin, host names of new clients are passed to the twin on the client's first lock request. Thus, the rpc.statd would be the one passing state information to the twin's rpc.statd. Recall that the rpc.statd is contacted by the rpc.lockd on the first lock request of a client. The rpc.statd is told to monitor that client. The rpc.statd in turn creates a file in the directory /etc/sm. This file name matches the host name of the client making the request. With this procedure the rpc.statd can then recover the list of clients that were being monitored before failure. After the rpc.statd has placed the file named after the client in the /etc/sm directory, it will contact the twin's rpc.statd. The twin's rpc.statd also places an entry in the /etc/sm directory that corresponds to the primary server's entry.

With the corresponding /etc/sm entries in place on the HA-NFS twin, all it has to do during takeover is to go through the lock recovery protocol playing the role of a recovering server for both itself and its twin. The clients of the failed server and those of the twin will execute lock recovery and the twin will effectively rebuild the locking state held by the primary server.

5. Our Design

The third design alternative involves the least overhead during normal failure-free operation. Tables 1 and 2 show the details of this protocol. Table 1 shows how a lock is obtained. Table 2 explains how locking state is re-established at the backup after a server fails. This design requires that the rpc.lockd be stopped and restarted during the takeover process to force the lock recovery to occur. One disadvantage of this scheme is that locking state has to be rebuilt instead of being already available as in the second scheme or getting it from the log as in the first scheme. We considered this trade-off acceptable since we were getting better performance in the normal case for sacrificing some performance during recovery from failure.

For simplicity in the chosen design, we decided to let the rpc.lockd reclaim lock state for itself in addition to the failed server. It would be possible to modify the rpc.lockd so that it would not drop its own locking state during recovery of the twin HA-NFS server's locking state.

- An application requests a lock on a file that resides in an NFS file system.
- The NFS client's kernel makes a RPC to the client's rpc.lockd requesting the lock
- If this is the first lock request for the server, the rpc.lockd on the client registers the server's host name with the rpc.statd on the client.
- The client's rpc.lockd sends the lock request to the server's rpc.lockd.
- If the lock request is the first one received from this particular client, the rpc.lockd registers the client's host name with the rpc.statd on the server.
- The server's rpc.statd then informs its twin rpc.statd on the backup server that the client should be monitored and then sends an acknowledgement to the rpc.lockd.
- The server's rpc.lockd makes the lock request to the server's kernel.
- The server's kernel accepts the lock request and validates it. Returns response to the server's rpc.lockd.
- The server's rpc.lockd responds to the client's rpc.lockd.
- The rpc.lockd on the client responds to the NFS client's kernel with the lock response.
- The application is given the answer to its lock request.

Table 1: Getting a lock on remote file in HA-NFS

The rpc.statd on a system is informed of the name of its twin host through a RPC call by a HA-NFS daemon when the HA-NFS subsystem is started. Once this is done, the rpc.statd will contact the twin's rpc.statd every time it is called by the local rpc.lockd with a new host to be monitored. The twin's rpc.statd will create an entry in the /etc/sm directory with the host name specified by the primary's rpc.statd and respond to the monitoring request.

If a server fails, the HA-NFS daemons running on its twin will detect the failure and take over its disks. They will then replay the log and bring the file systems to a consistent state and mount them on the appropriate directory. Finally, they will take over the IP address of the failed server on a spare network interface provided for this purpose. The network interface may be either ethernet or token ring. Finally the rpc.lockd is restarted. Because of the restart, the rpc.lockd and rpc.statd will go through the lock recovery protocol sending out

- The failure of the twin server is detected by the backup HA-NFS server. The backup server takes over the disks, brings the file systems to a consistent state and rebuilds the duplicate cache of the failed server. The rpc.lockd is stopped to prevent requests from being processed until takeover is complete. The backup then takes over the IP address of the failed server and starts to provide NFS file service.
- The rpc.lockd is restarted at the backup server. When it starts, it contacts the server's rpc.statd to tell it of its failure.
- Upon receiving the failure notification from the rpc.lockd, the rpc.statd at the backup server contacts each of the clients (both its own clients as well as those of the failed twin) that were being monitored because of the current locking state. This notification lets each of the clients know that the server's rpc.lockd has failed. The rpc.statd notifies each of the clients playing the role of the appropriate server.
- After the rpc.lockd notifies the rpc.statd of its failure it goes into a grace period for lock recovery. This allows clients to reclaim locks that they held before the failure of the twin servers rpc.lockd. The default grace period is 45 seconds.
- When the client's rpc.statd receives the notification that the server has failed, it ''calls back'' to the rpc.lockd on the client to notify it of the failure of the server.
- When notified, the client's rpc.lockd will send reclaim requests for all locks currently being held that are from the failed server. These reclaim requests will be honored at the server. As a result the server will regain the locking state that was held prior to its failure.
- During the grace period on the server, the rpc.lockd will only honor *reclaim* requests from clients. This assures consistency for the clients that held locks prior to the failure. The locks will be held again when the server restarts normal lock service. Regular locking requests that the server's rpc.lockd receives will be returned to the requesting client with a message that the client should retry the lock request. After the grace period elapses, new lock requests start being honored.

Table 2: Lock Recovery After Server Failure in HA-NFS

requests to clients of both this server and its failed twin to perform lock re-claim actions. The clients will see a simultaneous failure of both the twin servers and send out reclaim requests to both. All these requests will be received by the operational twin (since it is responding to both IP addresses) and the correct locking state will be recovered.

Another detail of the rpc.statd modifications deals with contacting the clients upon server failure. The clients are notified of the server failure by receipt of an RPC. Within the parameters of the RPC is the host name of the server that has failed. The rpc.statd at the client uses that host name to check if it among the list that is currently being monitored. If so, the rpc.statd then sends an RPC to the rpc.lockd at the client signifying the server failure. A simple solution would be for the twin rpc.lockd upon takeover to contact each client with the name of the failed server as well as its own host name. To get around this overhead, the file with the client's name in the /etc/sm directory indicates whether it is this server that needs to monitor the client or its twin.

A call to unmonitor a host needs to be supported for the twin-registration process. This is needed so that when a primary's rpc.lockd decides it no longer needs to monitor a client, the rpc.statd on both the primary and twin systems will remove the monitoring entry from the /etc/sm directory. When the rpc.statd on the primary HA-NFS server receives an unmonitor request from the rpc.lockd it will remove its /etc/sm entry for that host. The rpc.statd will then call the twin's rpc.statd with the request to remove the host from its monitoring tables. The rpc.statd on the twin will decrement the reference count of that monitored host. If the reference count is zero, it will remove its entry from the /etc/sm directory.

6. Reintegration

When a failed server recovers, the HA-NFS daemon at that server will recover the duplicate cache state from its twin after taking over the file systems.

The failed server will also receive the rpc.statd state from the twin. The mechanism to handle this is achieved through the twin registration process at the twin. After the duplicate cache state is transferred to the recovering server, the host name of the recovering server is registered with the twin's rpc.statd. By design the rpc.statd will transfer to the registered server the full list of host names that it is currently monitoring. This allows the recovering server to obtain the current list of clients that are being monitored.

After receiving the list of clients, the recovering server then restarts the rpc.lockd. Through its normal recovery process each of the clients in the transferred monitoring list are contacted and told of the failure of the recovering server. The rpc.statd at the notified client will then follow the normal lock recovery process by contacting the client's rpc.lockd allowing it to reclaim the client's locks.

The twin at this point will restart its rpc.lockd and it will also have the locking state rebuilt when the client's reclaim their locks. The rpc.lockd on the twin was originally stopped so that the file systems of the recovering system could be unmounted. When the rpc.lockd exits gracefully, the locks that it holds are released from the file system therefore freeing the file systems of reference counts that may prevent them from being unmounted.

7. Evaluation

The effort it took to implement the design chosen was minimal. A simple RPC program was designed to handle the twin registration and the transfer of host monitoring requests between rpc.statd's. The rpc.statd code was structured in such a way that the extra logic required to implement our design was small. Most of our effort was spent on understanding the design and implementation of the rpc.lockd and rpc.statd prior to modification. After the design of these two daemons was understood it was straightforward to design and implement the method chosen. There are three areas where the performance of a HA-NFS server and its clients will be affected by our design for the highly available lock manager. This performance penalty is in comparison to a standard NFS server and the standard implementation of the network lock manager protocol.

1. The first performance penalty is taken when a given client makes the very first lock request of the HA-NFS server. Contacting the twin server for the monitoring of the client will add extra delay in responding to the client. No penalty is incurred for subsequent lock requests.
2. The second penalty will be paid when a twin fails and the twin that takes over its identity starts to process the lock requests of its own clients and the clients of the failed twin.
3. The third penalty is incurred when the reintegration of the failed twin occurs. The twin server that has taken over must transfer the monitoring state to the recovering twin. In this case the implementation has the rpc.statd forking a child that handles the transfer of the monitoring state.

Because the first two penalties are more important, they will be discussed in further detail.

Penalty For The First Lock Request

In both the standard NFS and our lock manager designs, the first time a client makes a lock request, the rpc.lockd contacts the rpc.statd, and asks that the client's name be placed in the /etc/sm directory.

This is done by creating a file name which corresponds to the host name of the client. In our design, the rpc.statd at this point of first registration will also contact the twin's rpc.statd and have it monitor the same client. The rpc.statd makes this RPC to the twin's rpc.statd and waits for a response before responding to its rpc.lockd monitoring request. This means that the rpc.lockd will not be able to continue processing the lock request of the client until the rpc.statd at the twin finishes creating its file that corresponds to the initiating twin.

This extra overhead of contacting the twin's rpc.statd will obviously delay the response to the first lock request of the client. Table 3 illustrates the impact of this delay. A configuration of Risc Systems/6000s were used to measure what the cost of this first lock request would be in a HA-NFS environment. Three systems were used for these measurements. They were running AIXv3 with the HA-NFS subsystem installed and configured. The rpc.lockd and rpc.statd had been modified to follow our design. The three systems were isolated on a 16 Mbit/s token ring. The test case that was executed reset the systems so that no previous rpc.statd state was held at any system (client or server). The test case at the client mounted one of the HA-NFS exported file systems from the HA-NFS server pair. The test case then noted the current time and issued a system call to obtain a lock on a file in the NFS mounted directory. After the lock system call returned, the current time again was noted and the elapsed time measured. This is reported on the row labelled "First lock". The test case went on to do the same sequence of lock and unlock requests a second time. This second iteration however did not reset the rpc.statd state. Therefore the overhead of obtaining a second lock was measured. This is reported on the row labelled "Second lock".

This test case was also executed with a standard NFS server for comparison. The same configuration described above was used except that it was just one standard NFS server and one client. This time the rpc.lockd and rpc.statd were running the standard algorithm. Again the rpc.statd state on both client and server was removed and the test case executed. The second lock request was also executed as in the scenario described above.

Both of these lock test scenarios were executed 50 times and an average response time for the lock request and standard deviation calculated. These are the results reported in Table 3.

The overhead of contacting the twin server to have the rpc.statd monitor the client almost doubles the response time for the very first lock request. We felt that this cost was reasonable given that it occured only for the first lock request from a particular client. The numbers for the "second lock" request with and without HA-NFS are the same within the limits of experimental error.

The same was done for the unlock request and again the elapsed time reported. Table 4 shows the measurements for the unlock requests. This data shows that HA-NFS unlock requests do not suffer from any extra overhead.

	Without HA-NFS		With HA-NFS	
	Lock Oprn	Std. Dev.	Lock Oprn	Std. Dev.
First Lock	132.12 ms	13.28 ms	245.06 ms	70.18 ms
Second Lock	15.66 ms	0.98 ms	16.08 ms	1.20 ms

Table 3: Highly Available Lock Manager Overheads for locks

	Without HA-NFS		With HA-NFS	
	Unlock Oprn	Std. Dev.	Unlock Oprn	Std. Dev.
First Unlock	14.30 ms	0.41 ms	14.42 ms	0.80 ms
Second Unlock	14.43 ms	0.47 ms	14.63 ms	0.60 ms

Table 4: Highly Available Lock Manager Overheads for unlocks

	Without HA-NFS		With HA-NFS	
	Lock Oprn	Unlock Oprn	Lock Oprn	Unlock Oprn
First Run	14.80 ms	13.89 ms	15.19 ms	13.97 ms
Second Run	14.82 ms	13.85 ms	14.91 ms	13.90 ms
Third Run	14.80 ms	13.89 ms	15.17 ms	13.98 ms

Table 5: Highly Available Lock Manager Overheads (500 operations)

We believe that the typical mode that clients use servers in most applications is that a client would tend to get many locks on a particular server in a given period. This would tend to wash out the effect of the higher HA-NFS first lock overhead as compared to the standard NFS lock manager. Table 5 shows a test case that executed 500 sequential lock operations in the same configurations specified above (unlock operations were also measured). The results reported are the per request elapsed time.

Overhead Of Handling The Failure Of A Twin

The steps that are taken when a server fails and its twin takes over operation for the failed server have been enumerated in Table 2. Remember that one of the steps is to stop the rpc.lockd while the takeover configuration is executing on the twin. Once the takeover is complete, the rpc.lockd is restarted. At this point, the rpc.lockd will notify the rpc.statd of its failure and the rpc.statd will execute its lock recovery algorithm.

With our design, the rpc.statd will have to contact each of the clients twice. One RPC will contain the twin's host name and the second RPC will contain the host name of the failed server. In this way the clients will be notified of both server's failure and they will reclaim the locks held at the server pair. With our design the rpc.statd has exactly twice the normal number of RPC's to execute. This number may also include clients that held locks at the failed server and not at the twin that has taken over. Since this notification mechanism is asynchronous to the other operations occurring at the server it should not be a significant burden for the twin.

Also, the twin will have to handle its own incoming reclaim requests and the reclaim requests for the failed server. This load is difficult to determine since it depends on the type of applications that are being executed at the clients and their locking behavior. Therefore under heavy stress the default grace period that the rpc.lockd uses for lock reclaims may not be sufficient for correct operation. This is also true of a standard NFS server but is made worse by the fact that the twin will also be handling the failed server's lock requests.

In the testing that has been done with this implementation the recovery of client locking state was achieved in a reasonable amount of time. The majority of this testing was done under light to medium locking stress. Since the rpc.lockd has a relatively short default time for its grace period, the recovery process that the clients are forced to go through may fail under a very heavy load. If this happens, the grace period can be increased by the system administrator to handle this case.

It should be mentioned that the lock response time will not be the only thing that will suffer when a server fails. The normal NFS requests that the twin receives will also be affected by the extra work load that it has taken on as part of the impersonation of the failed server.

8. Network and Media Failures

HA-NFS provides recovery from disk and network failures for the file server as described in [3]. The same methods can ensure that the lock manager can recover from these failures.

Fast recovery from disk failures is achieved in HA-NFS by mirroring files on different disks. However, all copies of the same file are on disks that are controlled by the same file server, eliminating the overhead of ensuring consistency and coherence between the two servers that would otherwise occur. Since disk failures are not frequent, mirroring is only used for applications that require continuous availability. Otherwise, archival backups could be used to recover from disk failures. The files used by the rpc.statd could be mirrored to provide high availability.

Network failures are tolerated by optional replication of the network components, including the transmission medium. However, packets are not replicated over the two networks. Instead, the network load is distributed over the networks. Clients detect network failure because of loss of heartbeat from servers and switch over to the second network by changing their routing tables. Also, a message is sent to the server to change its routing tables. This mechanism works for all messages between client and server, including the locking protocol messages.

9. Comparison with Other Systems

Tandem's NonStop architecture [2] [5] uses special-purpose hardware in the form of dual-ported disk controllers which allow each disk to be attached to two processors. If a single processor fails, the other takes over the disks and provides processes that were using these disks with continued access. However, Tandem has the concept of process-pairs. Thus each I/O process has a twin to which it continuously checkpoints its state. This ensures that the backup I/O process knows what operations are needed to bring the disk to a consistent state when it takes over. On the other hand, HA-NFS has no such checkpointing overhead during normal (failure-free) operation. The information for bringing the disks to a consistent state is stored on the disk itself by treating each NFS client-to-server RPC as a transaction and writing a log. Thus, there is a significant difference between the HA-NFS and Tandem approaches. Presumably in the Tandem approach the same processor-pair/checkpointing approach is used to transfer locking state from one process to its backup twin. In our approach, the rpc.statd communicates with its twin rpc.statd only when a new client makes its first lock request. No communication is required for subsequent lock requests.

VAXcluster [6] also has a distributed lock manager which recovers locks after a processor fails. Upon being notified of node failure, the lock manager on each node must perform recovery actions before normal cluster operation continues. First, each lock manager deallocates all locks acquired on behalf of other processors. Only local locks are retained. Next, each lock manager acquires each lock it had before the failure. The net result is to deallocate all locks owned by the failed node. However, note that this requires *all* locks to be re-acquired on any failure. In NFS and HA-NFS, clients that held locks at a failed server node need to re-acquire only those locks that were held at the failed node. The trade-off is that the first lock request is slower in HA-NFS.

10. Future Work

We used a simple design where the twin of a failed server rebuilds both its own locking state along with the locking state of the failed server. The load of this server would decrease if only the failed server's locking state were to be selectively rebuilt. This can be done by having the rpc.statd keep a more detailed record of what clients were monitored by which server. This way only the clients affected by a takeover would be notified of the server failure.

The other part of the design that might be extended has to deal with the grace period that the rpc.lockd uses. The grace period is used to allow the clients to rebuild their locking state after a server failure. This grace period in the implementation is a default of 45 seconds. As mentioned earlier, this default seems to work well with the test cases used but it may not be sufficient when the load of the server increases. There is a possibility that the grace period could be dynamically decided based on the number of clients that respond to the failure message that the rpc.statd supplies. The rpc.statd could possibly keep track of the percentage of clients that have contacted the server after being notified of the failure. Once a certain percentage has been reached the rpc.statd could then notify the local rpc.lockd so that it can make a decision to either continue the grace period or extend it. It is possible that clients do not need to contact the server after failure of the rpc.lockd so the percentage to use in determining validity of the grace period would not be simple.

These future work items could possibly decrease the work load of the server and increase the likelihood of correct and the timely reclamation of locking state.

Bibliography

[1] AT&T System V Interface Definition.

[2] Joel Bartlett, A NonStop Kernel, In *Proceedings of the Eighth Symposium on Operating Systems Principles, Vol 15 No 5, Dec 1981.*

[3] Anupam Bhide, Elmootaz Elnozahy and Stephen Morgan, A Highly Available Network File Server. In *Proceedings of the Winter 1991 USENIX Conference*

[4] C. Juszczak, Improving the Performance and Correctness of an NFS Server. In *Proceedings of the 1988 USENIX Conference.*

[5] J. Katzman, A Fault-Tolerant Computing System. In *Proceedings of the Eleventh Hawaii International Conference on System Sciences, Jan 1978.*

[6] N. Kronenberg, H. Levy and W. Strecker, VAXclusters: A Closely-Coupled System. In *ACM Transactions on Computer Systems, Vol. 4, No. 2, May 1986.*

Author Information

Anupam Bhide is a research staff member at the IBM T. J. Watson Research Center. His current research interests include database systems, operating systems, fault-tolerance and racquetball. In addition to his work on fault-tolerance in network file systems, he has worked on fault-tolerance in parallel database machine, and high performance transaction processing. He graduated with a Ph.D. from the University of California-Berkeley in 1988. Previously, he has a B. Tech from I.I.T.-Bombay and a M.S. from University of Wisconsin-Madison. He can be reached at anupam@watson.ibm.com.

Spencer Shepler is currently a software engineer for IBM in Austin Texas. His interests include operating systems and distributed systems specifically distributed file systems. Spencer graduated in 1989 from Purdue University with a Master of Science degree in Computer Science. Since that time, he has been working with and partly responsible for NFS and its implementation in the AIX V3 operating system. Reach him electronically at shepler@netmail.austin.ibm.com.

#ifdef Considered Harmful, or Portability Experience With C News

Henry Spencer – Zoology Computer Systems, University of Toronto
Geoff Collyer – Software Tool & Die

ABSTRACT

We believe that a C programmer's impulse to use **#ifdef** in an attempt at portability is usually a mistake. Portability is generally the result of advance planning rather than trench warfare involving **#ifdef**. In the course of developing C News on different systems, we evolved various tactics for dealing with differences among systems without producing a welter of **#ifdefs** at points of difference. We discuss the alternatives to, and occasional proper use of, **#ifdef**.

Introduction

With UNIX running on many different computers, vaguely UNIX-like systems running on still more, and C running on practically everything, many people are suddenly finding it necessary to port C software from one machine to another. When differences among systems cause trouble, the usual first impulse is to write two different versions of the code—one per system—and use **#ifdef** to choose the appropriate one. This is usually a mistake.

Simple use of **#ifdef** works acceptably well when differences are localized and only two versions are present. Unfortunately, as software using this approach is ported to more and more systems, the **#ifdefs** proliferate, nest, and interlock. After a while, the result is usually an unreadable, unmaintainable mess. Portability without tears requires better advance planning.

When we wrote C News [Coll87a], we put a high priority on portability, since we ran several different systems ourselves, and expected that the software would eventually be used on many more. Planning for future adaptations saved us (and others) from trying to force changes into an uncooperative structure when we later encountered new systems. Porting C News generally involves writing a few small primitives. There have been surprises, but in the course of maintaining and improving the code and its portability, we insisted that the software remain readable and fixable. And we were not prepared to sacrifice performance, since one of C News's major virtues is that it is far faster than older news software. We evolved several tactics that should be widely applicable.

The Nature of the Problem

Consider what happens when **#ifdef** is used carelessly. The *first* **#ifdef** probably doesn't cause much trouble. Unfortunately, they breed. Worse, they nest, and tend to become more deeply nested with time. **#ifdefs** pile on top of **#ifdefs** as portability problems are repeatedly worked around rather than solved. The result is a tangled and often impenetrable web. Here's a noteworthy example from a popular newsreader.[1] See Figure 1. Observe that, not content with merely nesting **#ifdefs**, the author has **#ifdef** and ordinary **if** statements (plus the mysterious **IF** macros) *interweaving*. This makes the structure almost impossible to follow without going over it repeatedly, one case at a time.

Furthermore, given worst case elaboration and nesting (each **#ifdef** always has a matching **#else**), the number of alternative code paths doubles with each extra level of **#ifdef**. By the time the depth reaches 5 (not at all rare in the work of **#ifdef** enthusiasts), there are potentially 32 alternate code paths to consider. How many of those paths have been *tested?* Probably two or three. How many of the possible combinations even make sense? Often not very many. Figure 2 is another wonderful example, the Leaning Tower Of Hostnames. It's most unlikely that *anyone* understands this code any more. In such situations, maintenance is reduced to hit-or-miss patching. If you find and fix a bug, how many other branches does it need to be fixed on? If you discover a performance bottleneck and work out a way to fix it, will you have to apply the fix separately to each branch? Now envision what happens when hurried or careless maintainers *don't* apply their fixes in all the places where they are relevant.

Philosophical Aspects

The key step in avoiding such messes is to realize that *portability requires planning.* There is an abundance of bad examples to show that portability cannot be added onto or patched into unportable software. Many of the problems we discuss stem from the "never mind good, we want it next week"

[1] To quote from the old UNIX kernel: "you are not expected to understand this".

approach to software.

Even the best planning cannot anticipate all problems, but it is important to retain the emphasis on planning even into ongoing maintenance. When a new portability problem surfaces, it is important to step back and *think* about the problem and its solution. Is this a unique problem, or the harbinger of a whole new class of them? Usually it's the latter, which makes planning all the more crucial: how can the solution deal with all of them, not just the current one? Failure to think leads to the patch-upon-patch approach to portability, rapidly producing unreadable and unmaintainable code.

Once the problem (class) and the solution are understood, then and only then it is time to start work on the code. Typically this will mean re-implementing parts of it, not just hacking up the old code to work somehow. This highlights another issue: to revise the code, you must understand it... and that means not making an incomprehensible mess this time to interfere with maintenance next time.

All of this is typically more work than just hacking in a quick fix. Sometimes a quick fix may be necessary, or later thought may show that an earlier ''solution'' was really a quick fix and needs generalizing. In such cases, it is important to *go back and fix the kludges*. The time is not wasted; it is an investment in the future.

More generally, portability requires time and thought. Nobody gets everything right the first time; getting the code right means taking the time to think about what went wrong, decide what the mistakes were, and go back and fix them.

The alert reader may notice that almost all the remarks in this section could also be applied to achieving high performance, high reliability, etc., and that no specific boundary between development and maintenance was mentioned. We've really discussed how to achieve high-quality software. In our experience, this approach works; we can't imagine any other that would.

Portable Interfaces

Systems do, unfortunately, differ. It's often possible to avoid system-dependent areas well enough that the same code will run on all systems; we'll discuss that later. But sometimes multiple variants are inevitable. Even within the UNIX family, there are significant variations between systems.

```
void
cleanup_rc()
{
    register NG_NUM ngx;
    register NG_NUM bogosity = 0;

#ifdef VERBOSE
    IF(verbose)
        fputs("Checking out your .newsrc--hang on a second...\n",stdout)
          FLUSH;
    ELSE
#endif
#ifdef TERSE
        fputs("Checking .newsrc--hang on...\n",stdout) FLUSH;
#endif
    for (ngx = 0; ngx < nextrcline; ngx++) {
        if (toread[ngx] >= TR_UNSUB) {
            set_toread(ngx);            /* this may reset newsgroup */
                                        /* or declare it bogus */
        }
        if (toread[ngx] == TR_BOGUS)
            bogosity++;
    }
    for (ngx = nextrcline-1; ngx >= 0 && toread[ngx] == TR_BOGUS; ngx--)
        bogosity--;                     /* discount already moved ones */
    if (nextrcline > 5 && bogosity > nextrcline / 2) {
        fputs(
"It looks like the active file is messed up.  Contact your news administrator,\n\
",stdout);
        fputs(
"leave the \"bogus\" groups alone, and they may come back to normal.  Maybe.\n\
",stdout) FLUSH;
    }
#ifdef RELOCATE
    else if (bogosity) {
#ifdef VERBOSE
        IF(verbose)
            fputs("Moving bogus newsgroups to the end of your .newsrc.\n",
                stdout) FLUSH;
        ELSE
#endif
#ifdef TERSE
            fputs("Moving boguses to the end.\n",stdout) FLUSH;
#endif
        for (; ngx >= 0; ngx--) {
            if (toread[ngx] == TR_BOGUS)
                relocate_newsgroup(ngx,nextrcline-1);
        }
#ifdef DELBOGUS
reask_bogus:
        in_char("Delete bogus newsgroups? [ny] ", 'D');
        setdef(buf,"n");
#ifdef VERIFY
        printcmd();
#endif
        putchar('\n') FLUSH;
        if (*buf == 'h') {
#ifdef VERBOSE
            IF(verbose)
                fputs("\
Type y to delete bogus newsgroups.\n\
Type n or SP to leave them at the end in case they return.\n\
",stdout) FLUSH;
            ELSE
#endif
#ifdef TERSE
                fputs("y to delete, n to keep\n",stdout) FLUSH;
#endif
            goto reask_bogus;
        }
        else if (*buf == 'n' || *buf == 'q')
            ;
        else if (*buf == 'y') {
            while (toread[nextrcline-1] == TR_BOGUS && nextrcline > 0)
                --nextrcline;           /* real tough, huh? */
        }
        else {
            fputs(hforhelp,stdout) FLUSH;
            settle_down();
            goto reask_bogus;
        }
#endif
    }
#else
#ifdef VERBOSE
    IF(verbose)
        fputs("You should edit bogus newsgroups out of your .newsrc.\n",
            stdout) FLUSH;
    ELSE
#endif
#ifdef TERSE
        fputs("Edit boguses from .newsrc.\n",stdout) FLUSH;
#endif
#endif
    paranoid = FALSE;
}
```

Figure 1: Example of overuse of #ifdef

#ifdef, or something similar, ultimately is unavoidable. It can be *managed*, however, to minimize problems.

Among the basic principles of good software engineering are clean interfaces and information hiding: when faced with a decision that might change, hide it in one module, with a simple outside-world interface defined independently of exactly how the decision is made inside. One would think that well-educated modern programmers would not need to be taught the virtues of this technique. Unfortunately, **#ifdef** doesn't hide anything, and the interface it creates is arbitrarily complex and almost never documented.

The best method of managing system-specific variants is to follow those same basic principles: define a portable interface to suitably-chosen primitives, and then implement different variants of the primitives for different systems. The well-defined interface is the important part: the bulk of the software, including most of the complexity, can be written as a *single* version using that interface, and can be read and understood in portable terms. It is common wisdom[2] that localizing system dependencies in this way eases porting in cases where the code must actually be rewritten. Our point is that it makes the code simpler, cleaner, and more manageable even when no rewrite is expected.

[2]Common wisdom, n: something that is widely known but usually ignored. (UNIX programmer's definition.)

As a small case in point, when part of C News wishes to arrange that a file descriptor associated with a *stdio* stream be closed at *exec* time, to avoid passing it to unprepared children, this is done by

```
fclsexec(fp);
```

(where *fp* is the *stdio* structure pointer) rather than by some complex invocation of *ioctl* or something similar. Only the *implementation* of *fclsexec* needs to be cluttered with the details. (As others have noted in the past [ODel87a, Spen88a] in other contexts, one paradoxical *problem* of UNIX's not-too-complex system interfaces is that that they have discouraged the development of libraries with cleaner, higher-level interfaces.)

This confines **#ifdef**, but at first glance doesn't seem to eliminate it. Sometimes several system-specific primitives will be compiled from the same source, with portions selected by **#ifdef**. Note that even limiting the damage can be very important. However, in our experience, it's much more usual for the different variants to be completely different code, compiled from different source files—in essence, the parts *outside* the **#ifdef** disappear. The individual source files are generally small and comprehensible, since they implement *only* the primitives and are uncluttered with the complexities of the main-line logic. Out of 50 such source files in C

```
/* name of this site */
#ifdef GETHOSTNAME
      char *hostname;
#     undef SITENAME
#     define SITENAME hostname
#else /* !GETHOSTNAME */
#     ifdef DOUNAME
#           include <sys/utsname.h>
            struct utsname utsn;
#           undef SITENAME
#           define SITENAME utsn.nodename
#     else /* !DOUNAME */
#           ifdef PHOSTNAME
                  char *hostname;
#                 undef SITENAME
#                 define SITENAME hostname
#           else /* !PHOSTNAME */
#                 ifdef WHOAMI
#                       undef SITENAME
#                       define SITENAME sysname
#                 endif /* WHOAMI */
#           endif /* PHOSTNAME */
#     endif /* DOUNAME */
#endif /* GETHOSTNAME */
```

Figure 2: The Leaning Tower of Hostnames

News, half are less than 25 lines, most are under 50, and only a few are over 100. As an example, Figure 3 and Figure 4 are two implementations of *fclsexec*. There is hardly anything to be gained by trying to combine these two files into one file with **#ifdefs** every second line.

There are, of course, things that cannot conveniently be encapsulated as functions, for reasons of either interface or efficiency. But a ''primitive'' is not necessarily a function. Types and macros defined in a header file are also useful ways of hiding system-specific detail. Programmers often use such facilities on a small scale, e.g. the use of *off_t* as the system-supplied type for a size of a file or an offset within it, but they don't *write* such header files nearly as often as they should.

Although C's limited macro facilities hamper large-scale use of header-file encapsulation, more ambitious applications can be useful despite occasional clumsiness. As an example, consider our STRCHR primitive, which generates in-line code except on machines with compilers clever enough to do so automatically (see Figure 5). This is a bit awkward: what is being defined here is not exactly a function, but C preprocessor macros nevertheless force it to look like one. In the absence of a standard way to force inline expansion of normal functions, it remains a powerful technique for portable performance engineering despite its flaws: this and similar portable optimizations sped up major

```
/*
 * set close on exec (on UNIX)
 */

#include <stdio.h>
#include <sgtty.h>

void
fclsexec(fp)
FILE *fp;
{
    (void) ioctl(fileno(fp), FIOCLEX, (struct sgttyb *)NULL);
}
```

Figure 3: One implementation of *fclsexec*

```
/*
 * set close on exec (on System V)
 */

#include <stdio.h>
#include <fcntl.h>

void
fclsexec(fp)
FILE *fp;
{
    (void) fcntl(fileno(fp), F_SETFD, 1);
}
```

Figure 4: Another implementation of *fclsexec*

```
#ifdef FASTSTRCHR
#define STRCHR(src, chr, dest) (dest) = strchr(src, chr)
#else
#define STRCHR(src, chr, dest) \
    for ((dest) = (src); *(dest) != '\0' && *(dest) != (chr); ++(dest)) \
      ; \
    if (*(dest) == '\0') \
            (dest) = NULL           /* N.B.: missing semi-colon */
#endif
```

Figure 5: To inline or not to inline

components of C News by 40% without serious loss of clarity.

If one must use **#ifdef**, and it cannot be confined to header files and the like, one good rule of thumb is *use* **#ifdef** *only in declarations* (where ''declarations'' is understood to include macro definitions). This at least encourages some thought about defining an interface, rather than just hacking in something that somehow seems to work.

Finally, when defining interfaces, it is important to *document* them. The biggest reason for doing this is that it is important discipline that forces you to think about the issues and fill in fuzzy spots. The resulting documentation is also very valuable for maintenance. Perhaps somewhat surprisingly, it's also valuable for development, even if the project is not an army-of-ants operation using buildings full of people. We found it very important to document crucial interfaces like our configuration primitives, even though only two people were involved, to make sure things were being done consistently and we understood each other.[3]

Standard Interfaces

Of course, good interface design is not simple, especially given the limitations of existing programming languages. Often the best way to solve this problem is to avoid it instead. If an interface is needed, there is much to be said for choosing one that is already standard.

[3]Indeed, places where internal interfaces weren't completely documented were fruitful sources of misunderstandings, bugs, and a certain amount of snarling at each other.

There are several sources of reasonably decent standard interfaces, notably ANSI C [Inst89a] and POSIX 1003.1 [Engi90a]. Since these standards are quite recent, many of the systems of interest do not implement them fully. This doesn't preclude using the *interfaces*, however: you can supply your own implementation(s) for use on outdated systems. An example is the ANSI function *strerror* (shown in Figure 6).

This approach does impose a few constraints, since the standard interfaces sometimes are a bit ugly, and often aren't ideal for every program. It's tempting to come up with customized ones instead. But the standard ones have major advantages. For one thing, people understand (or will understand) them without having to decipher your code. For another, on systems which *do* implement the standard interfaces, the system-provided ones can be used. (This is particularly significant for primitives like *memcpy*, where system-specific tuning can produce major improvements in efficiency [Spen88a]. If you define your own customized interface, you must do your own customized implementation, which denies you the opportunity to benefit from the work of others.) For a third, while the standard interfaces may not be ideal, by and large they contain no grievous mistakes, and avoiding disasters is usually more important than achieving a precisely optimal solution. Finally, a standard interface saves endless puzzling, not to mention uncomplimentary speculation, by later maintainers: ''did he have some deep subtle reason for using a non-standard interface, or was he just stupid?''.

Reimplementing a standard interface can be a useful tactic when the standard interface does the right thing but the usual implementations perform

```
/*
 * strerror - map error number to descriptive string
 *
 * This version is obviously somewhat UNIX-specific.
 */
char *
strerror(errnum)
int errnum;
{
        extern int sys_nerr;
        extern char *sys_errlist[];

        if (errnum > 0 && errnum < sys_nerr)
                return(sys_errlist[errnum]);
        else if (errnum != 0)
                return("unknown error");
        else
                return("no details given");
}
```

Figure 6: strerror

poorly. A version which is faster but compatible can solve performance problems while leaving the door open to the possibility that the system implementations will improve someday. The *stdio* library is a particular case in point: old implementations of functions like *fgets* and *fread* are extremely inefficient, and even modern ones often can be improved on. This particular case gets tricky, because doing better means relying on ill-documented and somewhat variable internal interfaces,[4] but the performance wins for C News are so massive that we nevertheless did it.

Pitfalls that need careful attention when using standard interfaces are error checking and boundary conditions. It is important not to make assumptions that aren't in the standard. For example, a depressing amount of UNIX software assumes that *close* never returns any interesting status. Unfortunately, as networked file systems get more common and other complications are introduced, it is not at all unthinkable for an I/O error to be discovered only at *close* time. Meticulous error checking is important [Darw85a]. For example, see Figure 7.

Finally, note that standard interfaces exist on more than just the C level. By including an "override" directory early in the shell's search path, it becomes trivial to substitute reimplemented programs for standard ones that are missing or defective. We have a remarkably large—and steadily growing—list of known portability problems that arise from defective implementations of standard UNIX programs.[5]

[4]The standard build procedure for C News runs a test program to check compatibility with the local *stdio* implementation.

Inside-Out Interfaces

Sometimes there simply isn't any way to provide a necessary primitive on some systems. For example, most modern UNIXes permit setting the real userID to equal the effective userID, but some old systems allow only *root* to change the real IDs... and it is necessary to change the real IDs to create directories with proper ownerships. Given that many people will be[6] reluctant to let a large and complex program written by a stranger run as *root*, there doesn't seem to be any easy way out.

In this case, there is: turn the interface inside out, and have the dirty work done by caller rather than callee. Specifically, have the complex program invoked by a simple setuid-*root* program which sets things up properly on uncooperative systems.

[5]Lest anyone think we are disparaging porters and resellers only, we should comment that AT&T is as guilty as anyone else. For example, several releases of System V *make* have violated the System V Interface Definition in their handling of command lines like `test -s file` in *makefiles*. (Makefile command lines are specified to be executed just as if by the shell, but if *test* is a shell built-in and there is no actual *program* by that name, *make* often chokes and dies on this line.)

[6]Or *should* be!!!

```
/*
 * nfclose(stream) - flush the stream, fsync its file descriptor and
 * fclose the stream, checking for errors at all stages.  This dance
 * is needed to work around the lack of UNIX file system semantics
 * in Sun's NFS.  Returns EOF on error.
 */

#include <stdio.h>

int
nfclose(stream)
register FILE *stream;
{
    register int ret = 0;

    if (fflush(stream) == EOF)
        ret = EOF;
    if (fsync(fileno(stream)) < 0)      /* may get delayed error here */
        ret = EOF;
    if (fclose(stream) == EOF)
        ret = EOF;
    return ret;
}
```

Figure 7: Necessary error checking

A more mundane example is the problem of reading directories. Thanks to the lack of a library package for directory-reading in the oldest UNIXes, there isn't any standard way to do it. Raw reads don't work on 4.2+BSD systems (and increasingly-many others), the Berkeley directory library works well but has stupid name clashes with many old systems, and the POSIX library isn't widespread yet. Worse, because the insides of a directory-reading library are system-specific, it's difficult to provide a portable reimplementation of the POSIX functions.

The simplest way around this one is to move the problem out to a higher level of abstraction. The *ls* utility portably does the job, so wrap the invocation of the program in a shell file, with the list of names generated by *ls* and fed into the application as arguments or on standard input. The performance impact is rarely significant, and the alternative currently involves at least *six* different variants of the code, with more surfacing daily.

A less happy example of this technique is C News's *spacefor* program, used to check disk space so activity can be curtailed when it runs short. Its interface is simple and clean, and it is used everywhere in C News. Making it a shell program offered the possibility of exploiting the *df* command, which encapsulates the ugly complications of finding out how much space is available (and, sometimes, the *root* privileges needed to do so). Unfortunately, *df* is often relatively costly to invoke; worse, the only portable way to do 32-bit arithmetic from shell scripts is to use *awk*, which likewise tends to have considerable startup overhead. With some care, the performance impact was tolerable, although not entirely pleasant.

What we had not anticipated was that every little UNIX variant has its own different, incompatible *df* output format. Even "consider it standard" System V has at least three. The importance of program *output* being useful as program *input* [Ritc78a] has been disregarded completely. In the end, we found that while the *df* version remains useful—people with really odd systems can customize it easily—it was best to also provide C variants that use the three or four commonest space-determining system calls, improving (!) portability within a fairly large subset of UNIX variants.

Levels of Abstraction

In general, avoiding problems is better than solving them. The best way to solve portability problems is not to get involved with them. Sometimes they can't be avoided, but often a bit of ingenuity suffices to find a way around them.

The most powerful way of avoiding problems is to choose a level of abstraction where they don't show up. The *ls* example earlier was a case in point. The standard UNIX shell is a very powerful programming language, sufficiently removed from the lower levels of the system that shell programs are often highly portable. (Gratuitous differences in utility programs do get in the way, as do attempts to "improve" the shell that result in subtle or not-so-subtle incompatibilities, but this is usually a manageable problem.)

The usual objection to shell programming is the inefficiency of the result, but a careful division of labor between the shell and the programs it invokes is all that is needed. Most of the C News batching subsystem is written in shell, but it remains highly efficient, because most of its time is spent in the "`batcher | compress | uux`" pipeline, and those are all C programs.

Intermediate levels of abstraction, although harder to find, do exist. Substantial pieces of C News are coded in *awk* [Spen91a] where efficiency is not crucial and requirements permit.

One situation where high-level abstractions are particularly beneficial is when one must step outside "common base UNIX". Common base UNIX is essentially Version 7 [Labo82a], though the later V7 innovations have taken a while to find their way into System V (and some have never done so). POSIX 1003.1 [Engi90a] is mostly an attempt to codify common base UNIX. Unfortunately, common base UNIX did not address some issues at all, notably dealing with real-time networks like the Internet. Attempts to define interfaces for real-time networks [Divi83a, ATT86a] have generally resulted in complex and ugly messes.[7] Worse, there is no consensus on which one to use, and the quality of the designs can be judged by the rate at which they are being redesigned to deal with unexpected problems. Although higher-level abstractions for networking are not as common or as well-designed as they should be, networked file systems and shell invocation of programs like *rsh* can provide limited networking functionality without having to deal with the underlying mess.

A side benefit of high-level abstractions is that the resulting programs are generally far easier to modify and customize. This is a particularly important consideration for software intended to be run on many systems with varying administrative policies. Many system administrators who are not up to deciphering a 5000-line C program can cope quite well with modifying a 50-line shell script. We have made a conscious effort to put policy decisions in shell scripts, not in C code, wherever possible, and have had extensive and loud positive feedback on this.

[7]There are occasional exceptions like V10 Research UNIX [Cent90a] that are useful sources of interface ideas.

There is one negative aspect to moving to a higher level of abstraction: the resulting programs depend on a larger and perhaps more fragile set of underlying abstractions. Porting C News to a radically non-UNIX-like operating system reportedly typically involves little change to the C code, since the

```
#ifdef SYSLOG
#ifdef BSD_42
    openlog("nntpxfer", LOG_PID);
#else
    openlog("nntpxfer", LOG_PID, SYSLOG);
#endif
#endif

#ifdef DBM
    if (dbminit(HISTORY_FILE) < 0)
        {
#ifdef SYSLOG
        syslog(LOG_ERR,"couldn't open history file: %m");
#else
        perror("nntpxfer: couldn't open history file");
#endif
        exit(1);
        }
#endif
#ifdef NDBM
    if ((db = dbm_open(HISTORY_FILE, O_RDONLY, 0)) == NULL)
        {
#ifdef SYSLOG
        syslog(LOG_ERR,"couldn't open history file: %m");
#else
        perror("nntpxfer: couldn't open history file");
#endif
        exit(1);
        }
#endif
    if ((server = get_tcp_conn(argv[1],"nntp")) < 0)
        {
#ifdef SYSLOG
        syslog(LOG_ERR,"could not open socket: %m");
#else
        perror("nntpxfer: could not open socket");
#endif
        exit(1);
        }
    if ((rd_fp = fdopen(server,"r")) == (FILE *) 0){
#ifdef SYSLOG
        syslog(LOG_ERR,"could not fdopen socket: %m");
#else
        perror("nntpxfer: could not fdopen socket");
#endif
        exit(1);
        }

#ifdef SYSLOG
    syslog(LOG_DEBUG,"connected to nntp server at %s", argv[1]);
#endif
#ifdef DEBUG
    printf("connected to nntp server at %s\n", argv[1]);
#endif
    /*
    * ok, at this point we're connected to the nntp daemon
    * at the distant host.
    */
```

Figure 8: A truly awful style

UNIX and C programming interfaces are widespread even on non-UNIX systems, but substantial shell files relying on dozens of major UNIX utilities are more of a challenge. There is also the problem, mentioned earlier, of UNIX suppliers breaking formerly-working utilities.

Low-Level Portability

We assume that everyone reading this has had exposure to elementary notions of portability like using *typedef* names, avoiding stupid assumptions about the sizes of integers and/or pointers, being careful about byte order in interchange formats, etc. There are nevertheless a good many fine points that deserve some illumination, particularly in the area of how to use **#ifdef** safely.

As mentioned earlier, if **#ifdef** is needed at all, it is best confined to declarations, to try to preserve some explicit notion of interfaces. Such declarations, in turn, preferably should be confined to header (**.h**) files, to minimize the temptation to introduce **#ifdef** into main-line code.

An optional feature such as debugging assistance or logging can be defined as a macro or function that does nothing when not needed, else the full-blown function can be defined (perhaps in one of several system-specific ways, e.g. using a *syslog* daemon or not). At worst, this requires one **#ifdef** per such feature rather than the now-notorious style, seen in various bits of popular software, of clustering **#ifdef**s at the site of each *call* of said function(s), see Figure 8.

One awkward area[8] is functions with variable numbers of arguments. There is no way to write a C macro that can take a variable number of arguments, which makes it awkward to provide such an interface while still being able to hide the innards. Various tricks are in use, none of them entirely satisfactory; perhaps the least objectionable is an extra level of parentheses:

```
DEBUG(("oops: %s %d\n", b, c));
```

which lets a header file decide to either pass or discard the whole argument list:

```
#ifdef NDEBUG
#   define   DEBUG(list)   /* nothing */
#else
#   define   DEBUG(list)   printf list
#endif
```

A related problem is that definition of a variable-arguments function pretty well invariably involves some **#ifdef**ing to cope with the unfortunate differences between ANSI C *stdarg.h* and the traditional (although less portable) *varargs.h*.

[8]Actually, it's awkward in a great many ways, this being only one.

Although macros cannot take variable numbers of arguments, it *is* still possible to have them pick and choose among a fixed number of arguments. For example, the VERBOSE-TERSE business in one of our first exhibits, an attempt to avoid compiling in unneeded strings, can be handled with a macro:

```
MSG(short_form, long_form, iostream)
```

A short-form-only definition of the macro simply doesn't use the *long_form* argument. The choice can even be made at run time using *if* or the '?' operator, all by changing only the *definition* of the macro.

One valid use of **#ifdef**, particularly in header files, is the idiom

```
#ifndef COPYSIZE
#define COPYSIZE 8192
        /* unit of copying */
#endif
```

to supply a default value that can be overridden at compilation time (with **cc -DCOPYSIZE=4096**). One could wish for a shorter form (e.g., **#ifndefdef**), or even a compiler option allowing one to specify a value that overrides the first one defined in the program, since this idiom is common and very useful.

However, the first question to ask about such numeric parameters is whether they should be there at all. Consider:

```
#ifdef pdp11
#define LBUFLEN 512
        /* line buffer length */
#else
#define LBUFLEN 1024
        /* line buffer length */
#endif
```

This code presumes that people on small machines (or at least PDP-11s) prefer their programs to crash earlier than people on large machines. Any code using such (unchecked) fixed-sized buffers is prone to falling over and dying (or at best mysteriously truncating or wrapping long lines) anyway; the **#ifdef**s tip us off that these limits should be abolished and replaced with code that deals with dynamically-sized strings.

Another legitimate use of **#ifdef**, in fact required by the ANSI C standard in standard header files, is in protecting header files against multiple inclusion. In complex programs it can be quite difficult to ensure that a needed header file is included once and only once, and including it more than once typically causes problems with duplicate *typedef*s, structure tags, etc. Ignoring some issues of name-space control, the usual idiom for defending header files against multiple inclusion goes something like this:

```
#ifndef FOO_H
#define FOO_H 1
/* interface to the foo module */
```

```
typedef struct {
        char *foo_a;
        char *foo_b;
} foo;
extern foo *mkfoo();
extern int rmfoo();
#endif
```

(Some compiler implementors have invented bizarre special-purpose constructs, typically using ANSI C's **#pragma**, to avoid having the compiler re-scan the header file on later inclusions. That is not necessary. It suffices to have the compiler remember that the entire text of the file is inside the **#ifndef**, and hence need not be rescanned if `FOO_H` is still defined.)

```
#ifdef vax
    f(*ptr);
#endif
#ifdef pyr
    /*
     * darned Pyramid is so picky
     * about null pointers
     */
    if (ptr != NULL)
        f(*ptr);
#endif
#ifdef sparc
    /* the Sun 4 is just as bad! */
    if (ptr != NULL)
        f(*ptr);
#endif
/* ... */
```

Figure 9: Protecting broken code

#ifdef is often used to protect broken code in the style shown in Figure 9. The solution here is to face realities and write the code in a correct and portable manner:

```
/* avoid dereferencing null */
if (ptr != NULL)
        f(*ptr);
```

A related point, also illustrated by that example, is that if one *must* use **#ifdef**, one should test for specific features or characteristics (typically indicated to the compiler by symbols defined in a header file or on a command line), not for specific *machines*. There will almost always be another machine with the same problem. Consider the interesting bit of code shown in Figure 10. Rather mysterious, isn't it? What is so odd about Crays, and is it only Crays that are affected?

If testing for particular machines is unavoidable, perhaps because of some highly machine-specific operation, consider what happens if no machine is specified (or if the machine is one you've never heard of and hence didn't bother to list). Don't assume there is a default machine. It is much kinder to produce a syntax error than silently inappropriate code.

Occurrences of **#include** inside **#ifdef** should always be viewed with suspicion. There are better ways. Consider:

```
#ifdef NDIR
#ifdef M_XENIX
#include <sys/ndir.h>
#else
#include <ndir.h>
#endif
#else
#include <sys/dir.h>
#endif
#ifdef USG
#include <time.h>
#else
#include <sys/time.h>
#endif
```

This clutter could be avoided via judicious use of *cc -I/usr/include/sys* and consistent use of *dirent.h*, providing a fake one if necessary:

```
#include <direct.h>
#define dirent direct
```

Arranging, typically via a *makefile*, to put an "override" directory in the search path for header files is a tremendously powerful way of fixing botches in a site's header files *without* **#ifdef**.

When one uses **#ifdef**, one should base the tests on individual features:

```
#include <signal.h>
    /* may define SIGTSTP */

#ifdef SIGTSTP
  (void) signal(SIGTSTP, SIG_IGN);
        /* no suspension, thanks */
#endif
```

and not on (inaccurate) generalisations:

```
#ifdef cray
    } while (*s != '\r');                    /* till a newline (not echoed) */
#else
    } while (*s != '\n');                    /* till a newline (not echoed) */
#endif
```

Figure 10: Mysterious code

```
#ifndef SYSV
   (void) signal(SIGTSTP, SIG_IGN);
          /* no suspension, thanks */
#endif
```

or this example of the reverse problem (generalising from the specific) from a newsreader

```
/* Things we can figure out */

#ifdef SIGTSTP
#   define BERKELEY
   /* include job control signals? */
#endif
```

This particular point is worth emphasizing: the UNIX world is *not* cleanly split into System V and 4BSD camps, particularly with the advent of System V Release 4. Hybrid UNIXes are the rule, not the exception, nowadays.

Pragmatic Aspects of Portability

In practice, one encounters all manner of breakage in vendor-supplied system software: compilers, utilities (notably the shell and *awk*), libraries, kernels. Optimizers may need to be turned off if they are broken.[9] Installers may have to pick up working commands from other sources (e.g. the Usenet group **comp.sources.unix** or the GNU [Founa] project). Sometimes it is worth supplying simple but correct versions of small things (e.g. library functions) when a large class of machines is known to have broken ones. We ultimately decided that we could not provide complete replacements, or even workarounds, for all potentially-broken system software. Sometimes the problems are horrific enough that the right response is not to contort one's code but to get the customers to complain about the breakage until it is fixed.

Given all these potential problems, it is important to *detect* breakage as well as avoiding it or coping with it. We think very highly of *regression tests*, prepackaged tests that exercise the basic functionality of the software and check that the results are correct. They are very useful during development, both for bug-hunting in new code[10] and for confidence testing before release.[11] Of equal importance, though, is that they give the installer reasonable confidence that the software is actually working on his system, and that no glaring portability problems have escaped his notice.

Similarly, internal consistency checks, such as validated magic numbers in structures passed between user code and libraries, can save one's sanity by detecting breakage in system software early, before corruption spreads everywhere. Trying to debug a core dump by mail on an unfamiliar machine is not fun.

To a greater extent than we had anticipated, one learns about portability by porting. The system call variations among UNIX systems are fairly well documented and understood. The variations in commands were less well understood, at least by us, and the variations in programming environments were still more surprising. There is no substitute for *trying* your software on several seriously-different[12] machines before release. It's also worth making an effort to pick your beta-testers for maximum diversity of environments: we found a lot of unexpected problems that way.

Finally, a plea: if you find portability problems, *document them*. You can't expect everyone to actually *read* the documentation—we frequently respond to queries with "please read section so-and-so in document such-and-such, it'll tell you all about it"—but the more careful and conscientious installers benefit greatly from an advance look at known problems, especially when a truly weird system is involved.

Configuration

Given the senseless diversity in existing systems, some way to configure software for a new system is needed. Given that **#ifdef** can't do the whole job, how should we proceed? C News currently has an interactive *build* script that interrogates the installer about his system and then constructs a few shell scripts, which when run will use *make* to build the software. We intend to push most of the shell scripts into the *makefiles*, so that casual use of *make* works as people expect,[13] but the general approach seems to be a good one: ask which emulation routines and header files are necessary, rather than trying to guess. This strategy even allows cross-configuration and some degree of cross-compilation, which autoconfiguration schemes generally don't. It is also more trustworthy than autoconfiguration schemes, which can be fooled by some new innovation.

Almost all of *build*'s configuration questions[14] turn into choices of files rather than values for **#ifdef**

[9] The single most frequently reported "bug" in C News is actually a bug in a popular 386 C compiler's optimizer.

[10] One of us (HS) observes: "When I set up a regression test for software that has never had one before, I always find bugs. Always. *Every time.*"

[11] One very useful trick is to add a regression-test item looking for each bug that is found. This avoids the classic syndrome of having "fixed" bugs reappear in a later release.

[12] One problem we had: in our university environment, it was quite difficult to find System V machines. When we actually tried one, not long before our first real release, there were some unpleasant surprises.

[13] The main reason for not doing this from the start was the lack of a standard **#include** mechanism in *make*.

to examine. The few exceptions are mostly historical relics, and will be revised or deleted as time permits.

Statistics

A snapshot of current C News working sources shows 955 lines of header files and 19,762 lines of C files, split between 5,640 lines from libraries (including alternate versions of primitives), and 14,122 lines of mainline C code. Here is a breakdown of the **#ifdef** usage in that code:

reason	.h	main .c	dbz	rna	total
ifndefdef	13	40	6	0	59
comment	4	21	0	0	25
config.	6	25	19	7	57
protect .h	5	0	0	0	5
__STDC__	3	3	1	0	7
pdp11	2	0	0	0	2
lint	1	1	2	0	4
sccsid	0	1	0	0	1
STATS	0	5	0	0	5
other	0	1	0	0	1
total	34	97	28	7	166

The *.h* column represents header files. The *main .c* column represents all *.c* files other than those in the *dbz* and *rna* (Australian readnews) directories. The *ifndefdef* row represents the 'if not defined, define' idiom. The *comment* row represents uses of **#ifdef** to comment out obsolete, futuristic or otherwise unwanted code. The *config.* row represents uses of **#ifdef** to configure the software.

rna is presented separately because we inherited it rather than writing it. *dbz* is presented separately because it uses **#ifdef** heavily for configuration, for backward compatibility and to attempt to stand independently of C News. The main C files' use of **#ifdef** for ''configuration'' is misleading; in fact this is mostly vestigial code, superseded but not yet deleted from our current working copies.

Conclusions

Despite problems along the way, C News is outstandingly portable. It comes up easily on an amazing variety of UNIX systems. Other people have reported porting C News relatively easily to environments that we had considered too hostile, or at least too different from UNIX, to even consider as possible target systems: notably VMS, MS-DOS and Amiga DOS. The only major operating system known to present serious obstacles is VM/CMS.

In our experience, **#ifdef** is usually a bad idea (although we do use it in places). Its legitimate uses are fairly narrow, and it gets abused almost as badly as the notorious *goto* statement. Like the *goto*, the **#ifdef** often degrades modularity and readability (intentionally or not). Given some advance planning, there are better ways to be portable.

Acknowledgements

Thanks to Rob Kolstad for helpful comments on a draft of this paper. Thanks to James Clark for *grefer* (and the rest of *groff*). And thanks to the authors of our bad examples—you know who you are.

References

ATT86a. AT&T, *System V Interface Definition*, 2, 1986.

Cent90a. Computing Science Research Center, AT&T Bell Laboratories, Murray Hill, New Jersey, *UNIX Research System Programmer's Manual, Tenth Edition,* Saunders College Publishing, 1990.

Coll87a. Geoff Collyer and Henry Spencer, ''News Need Not Be Slow,'' *Proc. Winter Usenix Conf. Washington 1987*, pp. 181-190, January 1987.

Darw85a. Ian Darwin and Geoff Collyer, ''Can't Happen or /* NOTREACHED */ or Real Programs Dump Core,'' *Proc. Winter Usenix Conf. Dallas 1985*, pp. 136-151, January 1985.

Divi83a. Computer Science Division, Dept. of E.E. and C.S., UCB, *UNIX Programmer's Manual, 4.2 Berkeley Softare Distribution,* August, 1983.

Engi90a. Institute of Electrical and Electronics Engineers, *Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language] (IEEE Std 1003.1-1990) = ISO/IEC 9945-1:1990,* IEEE, New York, 1990.

Founa. Free Software Foundation, *GNU software,* anonymous ftp from prep.ai.mit.edu:/pub/gnu.

Inst89a. American National Standards Institute, X3J11 committee, *American National Standards Institute X3.159-1989 – Programming Language C, = ISO/IEC 9899:1990,* ANSI, New York, 1989.

Labo82a. Bell Laboratories, *UNIX Programmer's Manual,* Holt, Rinehart and Winston, 1982.

ODel87a. Mike O'Dell, ''UNIX: The World View,'' *Proc. Winter Usenix Conf. Washington 1987*, pp. 35-45, January 1987.

Ritc78a. D. M. Ritchie, ''UNIX Time-Sharing System: A Retrospective,'' *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1947-1969, 1978. Also in Proc. Hawaii International Conference on Systems Science, Honolulu, Hawaii, Jan. 1977.

[14] Of those that affect compilation at all; some questions are decisions affecting setup of control files for the compiled software to use.

Spen88a. Henry Spencer, ''How To Steal Code,'' *Proc. Winter Usenix Conf. Dallas 1988*, pp. 335-345, January 1988.

Spen91a. Henry Spencer, ''Awk As A Major Systems Programming Language,'' *Proc. Winter Usenix Conf. Dallas 1991*, pp. 137-143, January 1991.

Author Information

Henry Spencer is head of Zoology Computer Systems at the University of Toronto. He is known for his regular expression and string libraries, and as a co-author of the C News netnews software. Reach him via Canada Post at Zoology Computer Systems, 25 Harbord St., University of Toronto, Toronto, Ont. M5S 1A1 Canada. His electronic mail address is `utzoo!henry` or `henry@zoo.toronto.edu`.

Geoff Collyer leads C News development at Software Tool & Die. He is senior author of the C News netnews software. His interests include simple, small, fast, elegant and powerful system software. Reach him via U.S. Mail at Software Tool & Die, 1330 Beacon St. #215, Brookline, MA 02146. His electronic mail address is `world!geoff` or `geoff@world.std.com`.

Incl: A Tool to Analyze Include Files

Kiem-Phong Vo, Yih-Farn Chen – AT&T Bell Laboratories

ABSTRACT

Large C and C++ software projects typically share common types, macros, and variables among modules via include files organized into hierarchies. Many of these hierarchies grow so complex that it is hard for programmers to figure out when a file must be included. Since including unused symbols is usually harmless, application code tends to include more files than required. Knowing when files are or are not needed is useful to restructure the code and reduce the time required to build a product. It also helps in reorganizing the include hierarchies – should this be deemed necessary. *Incl* is a tool that analyzes include hierarchies to (1) show the dependencies among include files in graphical or text forms, (2) infer what files are not needed, and (3) provide ways to remove unused include files. The inference and removal of unused include files must be done with care for that may change the meaning of the application programs. We shall describe precise conditions under which include files can be safely ignored for compilation and give a linear time algorithm to compute such files. *Incl* has been used on many projects and experience shows that, in many cases, eliminating unnecessary include files significantly reduces compilation time.

Motivation

C and C++ software systems typically share data types, macros, and declarations of global variables by including common header files. The header files and their interdependencies form include hierarchies. As with any other parts of a software system, an include hierarchy grows with a project as features are added, deleted, or refined, and eventually becomes large and complex. For example, the include hierarchy for the X Window System graphics library and tools on our machine contains over a hundred files in several different directories.

When an include hierarchy is sufficiently complex, it is hard for programmers to find out exactly when a file must be included. Since including a file that does not contain useful information is usually harmless, the tendency is to include enough files so that the code will compile. For large projects, this practice may even be institutionalized by providing global header files that simply include the world. This practice simplifies programming at the extra cost of compilation overhead due to the processing of unneeded include files. For projects that distribute source code, long build time may convey to customers a poor image of quality. Therefore, at some stages of software development, it is useful to find out when an include file is needed or not needed. This information can be used to redo the code and avoid unnecessary include files. It also helps software architects to reengineer the include hierarchies – should that be necessary.

For a given source file, finding what set of include files is needed requires construction and analysis of complex reference relationships among symbols across the nested include files. As an example, Figure 1 shows the include hierarchy and reference graph of a typical X11 application program[1] In this picture, edges between files mean either inclusion or reference relationships:

- A dotted edge means that the tail file includes the head file but does not directly use any information contained in it.
- A dashed edge means that the tail file does not include the head file directly but refers to information contained in it.
- A solid edge means both inclusion and reference.

All include files that are unnecessary for the compilation of `load.c` are shown in ovals. Among the 22 include files, 12 are unnecessary. For example, `X11/Intrinsic.h` includes `X11/Object.h`, but does not use any symbols defined in that file (dotted edge). On the other hand, `X11/Object.h` does not include `X11/Intrinsic.h`, but refers to some symbols defined in it (dashed edge).

The exploration of relationships among include files is an example of software reverse engineering. One attempts to reconstruct high level information about large objects (in this case, files) in a software system from the source code. The C Information Abstractor [2] creates a C program database from C source files that stores, among other things, the reference relationships between all global program objects (types, macros, functions, variables, and files). To analyze relationships among include files, we need all of these reference relationships. In fact, the determination of when to exclude an include file must be done with care so that the meaning of the

[1]To simplify the picture, path prefixes of the form `/usr/include` were replaced with `UINC` and `/usr/local/include/X11` with `X11`.

program does not change. Based on the CIA[2] database, we shall describe precise conditions for an include file to become unnecessary during compilation and give a linear time algorithm to detect such files.

We implemented the include file analysis algorithm and many of its common applications in a tool, *incl*, which can be used to generate textual or graphical representations of relationships among include files. It can also generate scripts usable with other programs such as *ed* , the line editor, or *cpp*, the C preprocessor to exclude unneeded include files. The resulted tool has been used to reduce the compilation time by a third for many C source files. We shall give examples on the use of *incl* and statistics collected from a few projects.

[2]In this paper, we use *cia* to refer to the tool and CIA to refer to the system and concept.

Determining Unnecessary Include Files

The determination of what include files are needed depends on a number of factors defined by the compilation environment. This section discusses such factors and how they influence when files are needed. To simplify the discussion, for each file, we define the following sets based solely on the include relationships:

- *Closure(F)*: all include files included by F directly or indirectly.
- *First(F)*: include files directly included by F.

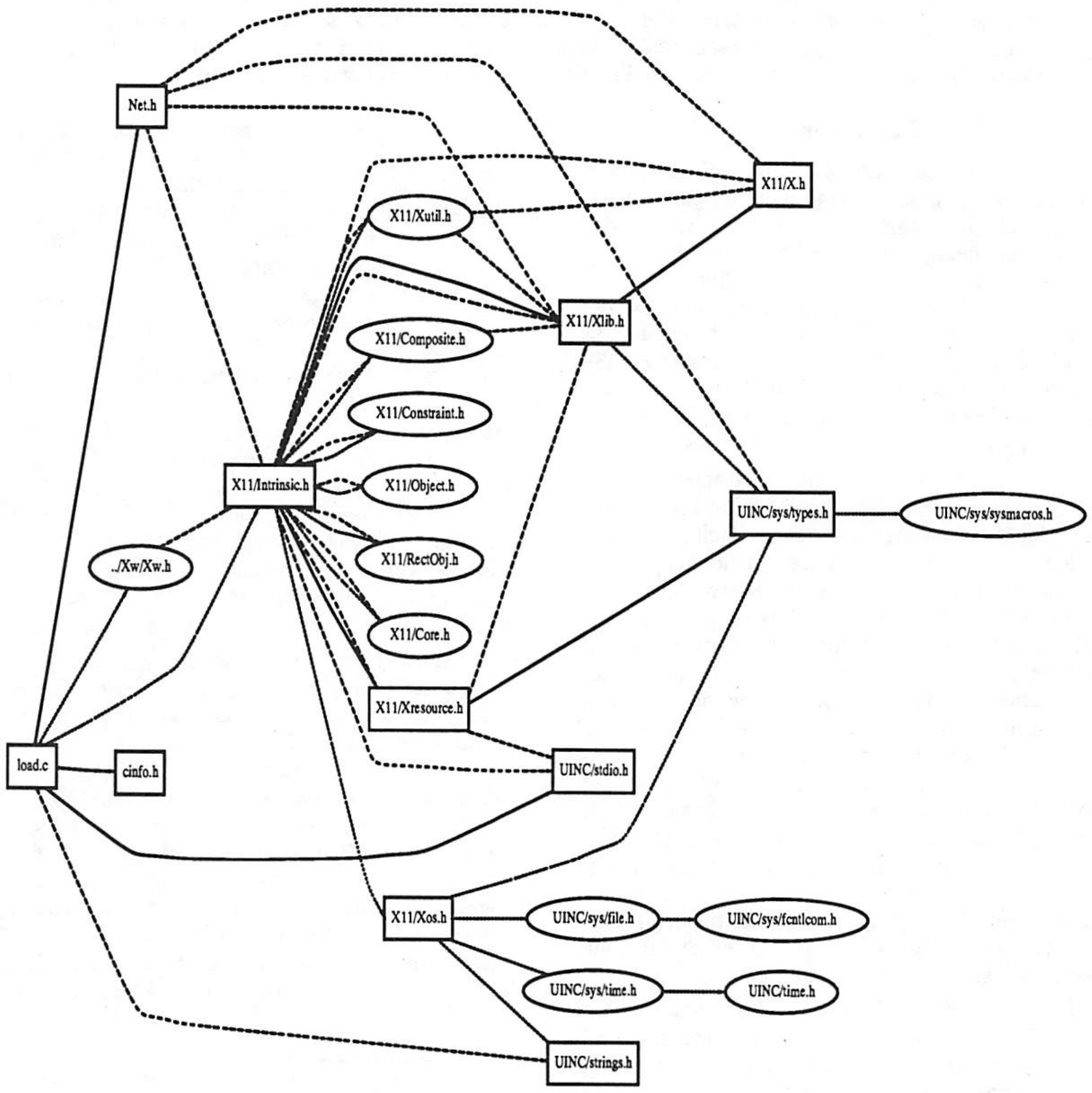


Figure 1: An Include and Reference Graph that Shows Unnecessary Include Files in Ovals

- *Nested(F)*: members in Closure(F) that are not in First(F).

For example, in Figure 1, First(`load.c`) consists of 5 members, Nested(`load.c`) consists of 17 members, and Closure(`load.c`) consists of 22 members.

C program global symbols can be divided into two classes, *definition* and *declaration*. A definition typically requires storage allocation while a declaration only gives type or value information without requiring any space in the generated code. Examples of definitions are functions and globally defined variables. Examples of declarations are macro and type definitions, or `extern` variables and functions. In the compilation of a C source file, all definitions are required. On the other hand, a declaration is required only if it is directly or indirectly referred to by some other definitions. If a declaration is not required, we shall call it *unused*.

The key observation about an unused declaration is that it cannot influence program behaviors. Therefore, reasonable compilation systems routinely ignore unused declarations in the generated code. This is the model of compilation that we shall assume for the rest of the paper. To rephrase, we assume that the generated code will remain the same with or without unused declarations. Given this, it would be nice if such declarations could be ignored completely during compilation. However, this is not always possible without extensive changes in either the source code or the underlying compilation system. Since virtually every C compilation system requires a file as the minimum compilation unit, a good compromise is to detect and ignore include files that consist only of unused declarations. This almost works except for two problems. The first problem is due to the way that most compilers process symbols. When a file is processed, all symbols that it refers to must be fully resolved whether or not they are really needed for code generation. This means that once a file is included, certain other files may be needed to resolve symbol references even though these symbols may not figure into the final code generation. The second problem is due to the way nested include files are processed. If an include file H is ignored, then all files in Closure(H) will also be ignored. This means that if an include file is deemed necessary by the above definition, at least a path of include files from the base file to it must be included. These considerations lead to the following recursive characterization of a necessary include file H:

1. H is necessary if it contains a definition of a variable or a function.
2. H is necessary if it contains a declaration of a symbol referenced by some other necessary file.
3. H may be necessary if it is on a path from the base file to a necessary file. If this path is unique, then H is necessary.

In developing an algorithm to mark files necessary, Conditions 1 and 2 are straightforward to implement. Condition 3 requires only that there is a path consisting of files marked necessary between the base file and any necessary file. Ideally, the number of marked files should be minimized. However, it is easily seen that this is an instance of the Steiner tree problem which is NP-complete [5]. Therefore, a heuristic approach is appropriate. The `path()` algorithm presented in the next section is a linear-time heuristic based on depth-first search.

```
[ hdr/db.h ]

#include "dir.h"
#include "cdb.h"
#include "error.h"

extern DIR *dbdir;

[ hdr/cdb.h ]

#define SYMDB "symbol.db"
typedef char *CDBNAME;
extern CDBNAME *dbs[];

extern DIR *dbdir;

[ hdr/error.h ]

#define ERR_FOPEN 1

[ hdr/dir.h ]

#include <sys/stat.h>
typedef char *DIR;
int CheckDir(/* char *dbdir */);

[ opendb.c ]

#include <stdio.h>
#include <ctype.h>
#include "db.h"

FILE *opendb(f)
char *f;
{
  FILE *fp;

#ifdef CDB
  if (!f) f=SYMDB; else f=dbs[0];
#endif
  if (!(fp=fopen(f, "w")))
    exit(ERR_FOPEN);
  else return(fp);
}
```

Figure 2: A Simple C Program that Includes Unnecessary Include Files

To illustrate the concepts of include and reference relationships, Figure 2 shows a small C source file and associated include files. This example will also be used later to demonstrate various uses of *incl*, the program to analyze include relationships among files. Figure 3 shows all include and reference relationships of this example, following the conventions used in Figure 1.

Following the definition of necessary files, we see that:

- Both `hdr/error.h` and `hdr/cdb.h` are necessary because there are symbols defined in these files referenced by `opendb.c`. They satisfy condition 2.
- `hdr/db.h` is necessary because it is on the include path from `opendb.c` to `hdr/cdb.h`. It satisfies condition 3.
- `hdr/dir.h` becomes necessary because it is referenced by `hdr/db.h`. It satisfies condition 2.
- `<ctype.h>` and `<sys/stat.h>` do not satisfy any of the three conditions; therefore, these two files are unnecessary for the compilation of `opendb.c`.

The Algorithm

Determining when files are needed for compilation requires knowledge of reference relationships among symbols. From the source code, the CIA system generates databases of global symbols and their reference relationships. We developed an algorithm to detect unnecessary include files based on CIA data and the definition given in the last section. Currently, CIA does not keep exact information about data structures and functions whose definitions span include files. This, in turn, incurs a limitation on the tool. However, one can easily argue that partial definitions of structures and functions in include files constitute bad programming practice. In our experiences, we have not seen code written this way.

To ease the description of the algorithm that detects necessary include files, we shall use a pseudo-C language. Each algorithm will be presented with line numbers which are used for references in subsequent discussions. First, we describe the primitives that retrieve data from a CIA database. These primitives are provided by CIA itself.

- `Ciaobj_t*    getciaobj(basefile)`: This function reads a CIA object record pertaining to the file `basefile`. An object record contains at least the following fields:
  - `name`: the name of the object.
  - `file`: the file where it is defined.
  - `sclass`: the storage class of the object. We are interested in whether or not the object is defined or just declared in the associated `file`.
- `Ciaref_t*    getciaref(basefile)`: This function reads a CIA reference record pertaining to the file `basefile`. Each reference record contains at least the following fields:
  - `name1, name2`: the names of the referring and referred objects.
  - `file1, file2`: the files where the objects appear.
  - `kind1, kind2`: the types of the objects.

The overall *incl* algorithm to detect necessary files is as follows:

```
 1: incl(basefile)
 2: {    root = makegraph(basefile);
 3:      dfnumber(root);
 4:      root->type = NECESSARY;
 5:      enqueue(root);
 6:      while(notempty())
 7:      {    node = dequeue();
 8:           reference(node);
 9:           path(node);
10:      }
11: }
```

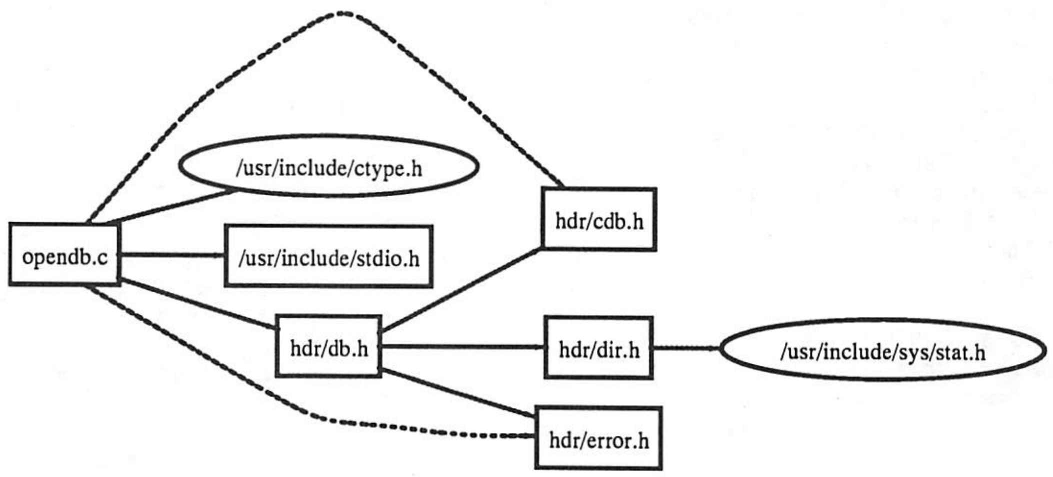


Figure 3: The Include and Reference Relationships in `opendb.c`

Remarks

Line 2 constructs the include/reference graph rooted at the file `basefile`. Line 3 computes a numbering of the nodes based on depth-first search. We shall omit the description of `dfnumber()` as its implementation is straightforward [1]. Lines 4-5 mark the root node as necessary, then insert it into the queue to be processed. We assume as given the operations `enqueue()`, `dequeue()` and `notempty()` which respectively inserts an element to the queue, deletes an element the queue and checks to see if the queue is not empty. Lines 6-10 process each element on the queue by first calling `reference()` to mark other necessary files that contain symbols directly or indirectly referenced from it. Then, `path()` is called to ensure that there is a path of files marked necessary from the root file to the given file.

Code Segment 1 shows the algorithm to construct the include/reference graph from a source file. Lines 2-10 read object descriptions from the CIA database and construct the nodes of the include graph. The function `getnode()` is called on line 3 to either find a node corresponding to the file `obj->file` or construct one if it does not yet exist. A node structure contains at least the following fields:

- `name`: the file name associated with the node.
- `type`: type of node.
- `edges`: edges outgoing from the node.
- `dfn`: the depth-first number of the node.

Lines 4-9 mark `node->type` as `NECESSARY` if the storage class of the object is statically defined (`s`) or globally defined (`g`) and insert the file into the queue of files that must be processed to resolve symbol references. This implements condition 1 of the definition of necessary files. Lines 11-22 read object reference information from the CIA database and construct the edges connecting the involved files. An edge connection is made via the function `getedge()`. Each edge structure contains the following fields:

- `node`: the head of the edge. This is the file being referenced.
- `type`: the type of edge.

Lines 15-21 mark `edge->type` as either `INCLUDE` if both objects are files or as `REFERENCE` otherwise. Note that an edge may be both `INCLUDE` and `REFERENCE`. When an `INCLUDE` edge is constructed, its opposite `INCLUDED` edge is also constructed. This opposite edge is used for an efficient implementation of the `path()` algorithm below. Finally, line 23 returns the root node of the include/reference graph.

Code Segment 2 shows the algorithm to mark files that contain symbols directly or indirectly referenced from a given file.

Line 2 iterates over each edge coming from `node`. Lines 3-4 ensures that only `REFERENCE` edges will be searched and only nodes that have not been searched will be searched. Lines 5-7 mark an

```
 1: makegraph(basefile)
 2: {   while((obj = getciaobj(basefile)) != EOF)
 3:     {   node = getnode(obj->file);
 4:         if(obj->sclass == 's' || obj->sclass == 'g')
 5:         {   if(node->type != NECESSARY)
 6:             {   node->type = NECESSARY;
 7:                 enqueue(node);
 8:             }
 9:         }
10:     }
11:     while((ref = getciaref(basefile)) != EOF)
12:     {   node1 = getnode(ref->file1);
13:         node2 = getnode(ref->file2);
14:         edge  = getedge(node1,node2);
15:         if(ref->kind1 == 'f' && ref->kind2 == 'f')
17:         {   edge->type |= INCLUDE;
18:             e = getedge(node2,node1);
19:             e->type |= INCLUDED;
20:         }
21:         else    edge->type |= REFERENCE;
22:     }
23:     return getnode(basefile);
24: }
```

Code Segment 1: Constructing include/reference graph unsearched node as `NECESSARY`, then recurse. `reference()` implements condition 2 in the characterization of necessary files.

Code Segment 3 is the algorithm to mark nodes on paths from the root file to a necessary files.

To understand how `path()` works, we note that `dfnumber()` does two things: (1) it computes a spanning tree rooted at the base file, and (2) it numbers the nodes as they are searched. Now, for any given node H, let *in* (H) be the set of nodes with edges pointing to H. A property of depth-first numbering is that, for all K in *in* (H), if the depth-first number of K is smaller than that of H, then K is an ancestor of H in the spanning tree. Further, such ancestor nodes are numbered so that smaller numbered nodes are ancestors of higher ones in the spanning tree. Given this, it is easy to see that the `for(;;)` loop between lines 3-12 examines and selects some node from the set of ancestors of `node` to mark necessary. Lines 7-8 stop the algorithm without marking any node if such a node already exists. Lines 9-10 make sure that if a new node is to be marked, the one that is closest to the root will be selected.

Let F be a source file to be compiled and H some include file in Closure(F). We note that each `NECESSARY` file H is enqueued once and dequeued once. When the file is dequeued, `path()` is called to ensure that there is another `NECESSARY` file closer to the base file on the spanning tree that directly includes H. Thus, an easy induction based on the distance from the root file shows that:

Theorem 1. If H is an include file of F that is marked `NECESSARY`, there is an include path from F to H in which all files are marked `NECESSARY`.

Next consider an include file H in Closure(F) that is not marked `NECESSARY` by `incl()`. We need to show that H can be safely excluded in the compilation of F. Assume by contradiction that there is some symbol s in H that may affect the compilation of F. If s is a defined symbol, then `makegraph()` would mark H as necessary, a contradiction. This means that s must be a declared symbol. In this case, s can affect the compilation of F only because it is referenced by some other symbol in a file X that is marked necessary. Since X is marked necessary, it must be enqueued at some point in time. Upon dequeuing, it is examined by `reference()`. Now, the check on line 5 of

```
 1: reference(node)
 2: {   for(edge in node->edges)
 3:     {   if(!(edge->type&REFERENCE) || edge->node->type == NECESSARY)
 4:             continue;
 5:         edge->node->type = NECESSARY;
 6:         enqueue(edge->node);
 7:         reference(edge->node);
 8:     }
 9: }
```

Code Segment 2: Marking referenced files

```
 1: path(node)
 2: {   mark = NULL;
 3:     for(edge in node->edges)
 4:     {   if(!(edge->type&INCLUDED))
 5:             continue;
 6:         if(edge->node->dfn < node->dfn)
 7:         {   if(edge->node->type == NECESSARY)
 8:                 return;
 9:             if(mark == NULL || e->node->dfn < mark->dfn)
10:                 mark = edge->node;
11:         }
12:     }
13:     if(mark != NULL)
14:     {   mark->type = NECESSARY;
15:         enqueue(mark);
16:     }
17: }
```

Code Segment 3: Marking nodes on paths reference() would cause H to be marked necessary, a similar contradiction. This proves:

Theorem 2. If H is an include file of F that is not marked necessary by incl(), then it is safe to exclude H in the compilation of F.

Theorems 1 and 2 show that incl() correctly implements the conditions for necessary include files discussed previously. Therefore, in an appropriate compilation model, the files that incl() does not mark as necessary can be safely excluded. It would be nice to have the reverse, i.e., all files marked necessary are required for compilation. However, this is not generally true. Consider an example where a base file a.c includes two files b.h and c.h. In turn, both b.h and c.h include d.h. Finally, c.h includes e.h. Suppose that neither b.h nor c.h contains any definitions or declarations required by a.c but d.h and e.h do. It is clear that, for correct compilation of a.c, only c.h, d.h, and e.h are required. However, path() may also mark b.h as necessary if dfnumber() searches b.h before c.h. This shows a limitation of the path() heuristic algorithm.

Finally, we need to analyze the time requirement of incl(). This can be divided into two parts: the construction of the include/reference graph and the search for necessary include files. In constructing the graph, the time is dominated by the primitives to access the CIA database. However, these primitives are called exactly once for each symbol and reference relationships. The CIA database is arranged so that it takes constant time to perform each call to these primitives. The search for an existing node or edge by the functions getnode() and getedge() can be implemented in hash tables so that each function call is constant time on average. This means that the graph construction phase can be done in linear time (on average). In the search for necessary include files, first note that the depth-first numbering of the nodes requires linear time. Then, note that each node in the graph can be marked as NECESSARY exactly once. Each REFERENCE edge is searched at most once when its tail node is examined by reference(). Likewise, each INCLUDED edge is searched at most once when its head node is examined by path(). Thus, the total run time for reference() and path() is linearly bounded in the number of files and relationships among them. To sum this up, we have:

Theorem 3. The algorithm incl() runs in linear time in the number of files, symbols and symbol references.

Analyzing Include Hierarchies

Section 3 shows that finding unnecessary include files is computationally feasible. The program *incl* simplifies the analysis of include hierarchies by providing a variety of ways to extract and present information. Using the C program shown in Figure 2 as a base, this section gives examples on the use of *incl*. We shall use the convention that a user input command line starts with $, the shell prompt. Any output from the command will immediately follow this line.

Before *incl* can be used, a C program database file opendb.A must be created with the *cia* command using the same options as given to the C compiler:

```
$ cia -c -Ihdr -DCDB opendb.c
```

Note that compiler options like -DCDB may influence the file include relationships because of #ifdef 's. This, in turn, influences the working of *incl*.

After the database opendb.A is created, *incl* can be used to generate the include and reference graph derived from opendb.c. The best way to see this information is to generate a picture such as the one in Figure 3 with:

```
$ incl -R7,3.5 opendb.A
$ dag -Tps opendb.d
```

Here, the option -R7,3.5 directs *incl* to generate a file opendb.d that contains a description of the include hierarchy as a directed graph to be drawn in an area that is 7 inches by 3.5 inches. Then, the program *dag* [4] is used to generate a picture specified in the PostScript language (the option -Tps).

Below is a quick scan to see what files are not needed. The result shows that two files can be skipped. One of them, (/usr/include/sys/stat.h), is included indirectly.

```
$ incl opendb.A
opendb.c:
    /usr/include/ctype.h
    /usr/include/sys/stat.h
```

To see the full hierarchy of include relationships with proper indentation, use the -l option. In this textual view, the character ~ tags unnecessary files in Closure(opendb.c):

```
$ incl -l opendb.A
opendb.c:
    /usr/include/stdio.h
    /usr/include/ctype.h
    hdr/db.h
        hdr/dir.h
            /usr/include/sys/stat.h~
        hdr/cdb.h
        hdr/error.h
```

After finding out what include files are not needed, a few options are available to eliminate or at least avoid them. The most direct approach is to edit a C source file F and remove any unneeded files in First(F). For example, the statement `#include <ctype.h>` can be safely deleted from `opendb.c`. However, deleting `#include <sys/stat.h>` from `hdr/dir.h` is dangerous because `hdr/dir.h` may be used by other programs. Let's suppose for now that we have control over `opendb.c` so we can delete any unnecessary `#include` statements from it. We can run *incl* with the option `-e` to generate an *ed* script, then run *ed* to actually delete the unnecessary statements:

```
$ incl -e opendb.A
opendb.c:
            /usr/include/ctype.h
$ cat opendb.e
2d
w
$ ed opendb.c < opendb.e
"opendb.c" 16 lines, 219 characters
#include "db.h"
"opendb.c" 15 lines, 200 characters
```

Though it is not generally safe to delete code from a source file, `incl -e` can help in software reengineering. An example of this is to partition a large module of code into smaller units. Each new unit may start by including all original include files. Then, `incl -e` can be used to modify the new units so that they will include only what is needed.

In contrast to source files, deleting code from include files is inherently dangerous because header files are usually shared. In certain development organizations, for a variety of reasons, programmers may not be allowed to delete code from certain source files. In such a case, we must rely on a smart compilation system to skip unnecessary include files. The C preprocessor developed by Glenn Fowler and distributed with *nmake* [3] takes a special option `-I-I-.u` that reads *file* `.u` to determine what files to skip during C preprocessing. Assuming that this special C preprocessor is available, `incl -u` can be used to generate appropriate `.u` files for compilation:

```
$ incl -u opendb.A
opendb.A:
$ cat opendb.u
"/usr/include/ctype.h"
"/usr/include/sys/stat.h"
$ cc -I-I-.u -Ihdr -DCDB -c opendb.c
```

Figure 4 shows the include graph of the file `graphics.c` from a picture drawing program. Most path prefixes of the filenames and reference-only edges were removed to simplify the picture. Out of the 67 files included directly or indirectly by `graphics.c`, there are 52 unnecessary files (shown in oval nodes). Using a similar compilation approach as in the above example, *cpp* skipped these 52 files, which reduced the total lines that cpp processed from 23155 to 9076 – a saving of 61%. The total compilation time of `graphics.c` went from 10.04 (user+sys) seconds to 6.64 seconds on a SUN Sparcstation I, a saving of 34%. We shall give more detailed statistics on three projects in the next section.

Statistics

To see the effectiveness of using *incl*, we experimented with compiling the source code from two different projects on a Solbourne running SUN OS 4.0. These projects are based on the graphics facilities provided in SunView and X respectively. Figure 5 shows the data from the experiment. For comparative purpose, the third column of Figure 5 shows data from a module in a large project on a different machine architecture. This data was given to us from a user of *incl*.

measure	*Project A*	*Project B*	*Project C*
`NumSrcFiles`	23	19	35
`NumIncFiles`	74	54	163
`NumIncScans`	1158	378	1168
`NumIncSkips`	821	209	479
`PercIncSave`	71%	55%	41%
`OrgCompTime`	152 secs	96 secs	15m58s
`SkipCompTime`	98 secs	85 secs	11m41s
`PercTimeSaved`	36%	12%	26%

Figure 5: Include File Statistics on Three Projects

Here are the measures displayed in the table:

- `NumSrcFiles`: the number of files in the database with suffix `.c`.
- `NumIncFiles`: the number of files with suffix `.h`, including all user and system header files.
- `NumIncScans`: the total number of times that include files are scanned. Note that an include file shared by several source files may be scanned several times.
- `NumIncSkips`: the number of include file scans that can be skipped with `incl -u`.
- `PercIncSave`: this is the ratio of `NumIncSkips` over `NumIncScans`.
- `OrgCompTime`: the time it takes to compile all `.c` files. The time taken is the sum of the user time and the system time obtained by using the UNIX regmark] *time* command (actually, a built-in shell command in our case). It does not include the linking time.
- `SkipCompTime`: the time it takes to compile all `.c` files by ignoring unnecessary include files.
- `PercTimeSaved`: one minus the ratio of `SkipCompTime` over `OrgCompTime`.

As Figure 5 shows, the compile time saving ranges from 12% to 36%. Note that there are two different wastage problems with processing header files: processing unnecessary files and processing files that are included multiple times. To avoid the latter, a number of the header files in the study are protected by pairs of `#ifndef, #endif`. This helps standard C preprocessors to avoid scanning such files more than once. Our C preprocessor (by Glenn Fowler) is smart enough to, in fact, avoid reopening such header files when applicable. Therefore,

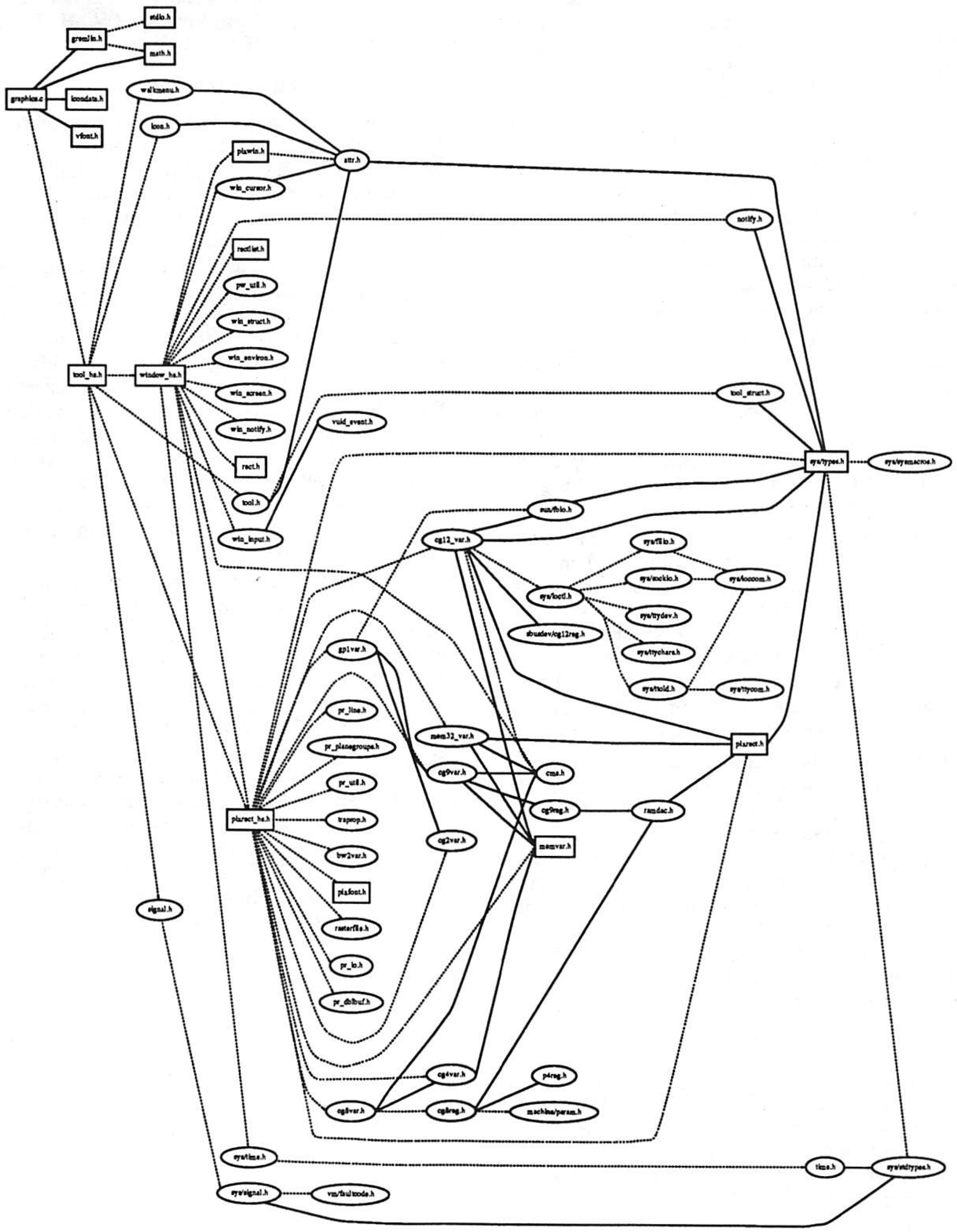


Figure 4: An Include Hierarchy with Many Unnecessary Include Files

the compile time via `incl -u` measures the true saving of skipping files that are unnecessary, and not just the saving of avoiding multiple file openings.

Conclusion

Reverse engineering is the activity of recovering from a system's implementation its high level objects and their interrelationships. This paper describes a tool, *incl*, suitable to analyze the relationships among include files in a C software system. The analysis of such files must be done with care if we want to use the resulting information to reengineer the software. We described a precise set of conditions under which include files could be safely ignored during compilation, implemented a linear time algorithm to compute such files, and proved the algorithm's correctness. Our method is general and could be used to analyze C++ include files, but we have not yet explored this direction.

Incl can be used to generate textual and graphical information on include hierarchies. Such information shows the include structures of large projects and provides a starting point in any effort to reengineer such structures. In a more limited fashion, *Incl* can also be used to generate scripts for automatic deletion of unused include files. We gave examples of how to use the program.

Eliminating unused include files can save significant compile time. *incl* can be used in conjunction with a smart C preprocessor to ignore unused include files during compilation. Note that this approach is different from the standard practice of using `#ifndef _HEADER_FILE` to avoid multiply included files. We presented experimental data showing that up to a third of compile time can be saved by ignoring unused include files.

Finally, *incl* is a part of the repertoire of C software analysis tools provided under the umbrella of the CIA system. It is a small application (650 lines of amply commented C code) written on top of the CIA database. The ease of its implementation shows that the CIA conceptual model and data provide a good basis for developing C analysis tools that deal with non-local C objects. This also shows the power of software tool modularity in which appropriate abstractions are captured and implemented in the right place.

References

[1] Alfred V. Aho and John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley Publishing Company, 1974.

[2] Yih-Farn Chen. The C Program Database and Its Applications. In *USENIX Baltimore 1989 Summer Conference Proceedings*, 1989.

[3] G. S. Fowler. A Case for make. *Software – Practice and Experience*, 20:35-46, June 1990.

[4] E. R. Gansner and S. C. North and K. P. Vo. DAG – A Program that Draws Directed Graphs. *Software: Practice and Experience*, 18(11), November 1988.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

Author Information

Phong Vo lost his B.A degree somewhere but did manage to retain an M.A and a Ph.D. in Mathematics from the University of California at San Diego in 1977 and 1981 respectively. He joined Bell Labs in 1981 and is currently a Distinguished Member of Technical Staff. His research interests include graph theory, data structures and algorithms, user interface and reusable software tools. Aside from obscure theoretical works, Phong is responsible or partially responsible for a number of popular software tools including the current System V <curses> and malloc libraries, IFS, the Interpretive Frame System, a language for building applications with menu and form interfaces, and DAG, a program to draw directed graphs. He was awarded an AT&T Bell Labs Fellowship this year. Phong can be reached at kpv@ulysses.att.com or Kiem-Phong Vo, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, New Jersey 07974.

Yih-Farn (Robin) Chen received the B.S. degree in electrical engineering from National Taiwan University, Taiwan, in 1980, the M.S. degree in Computer Science from University of Wisconsin, Madison, in 1983, and the Ph.D. degree in Computer Science from the University of California, Berkeley, in 1987. He is currently a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include the modeling and integration of software databases, programming environments, and network management. Yih-Farn can be reached at chen@ulysses.att.com or Yih-Farn Chen, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, New Jersey 07974.

Large Scale Porting through Parameterization

David Tilbrook, Russell Crook – Siemens Nixdorf Information Systems Ltd.

ABSTRACT

The advent of open systems and standards, while beneficial, has not eliminated the difficulty of maintaining and transporting large scale software systems across many varying platforms.

In this paper we discuss the need and criteria for a effective porting strategy, one that allows the rapid and inexpensive retargeting of large scale software systems to many widely varying platforms while not compromising the integrity of that software on any previously supported platform.

> ''*Getting Tigger down*'', said Eeyore, ''and *not hurting anybody.* Keep those two ideas in your head, Piglet, and you'll be all right.''
>
> A. A. Milne, *The World of Pooh*, 1957, pp216, McCelland & Stewart Ltd.

A key component of any porting strategy is the methodology used to determine, represent, use, and validate specifications of the target system's characteristics and site or system dependent build and run time controls. The standards efforts (e.g., POSIX, ANSI C) are attempting to eliminate the large number of discrepancies that exist among systems today. However, the problem will always exist, for reasons that are discussed.

Hence, the main objective of this paper is to present and justify the methodology that we use.

This methodology is in production use on several commercial products in Sietec. Its benefits include relieving the programmer from the burden of needing detailed knowledge of all the idiosyncrasies of the various target environments. It is sufficiently powerful that it accommodates many flavours of BSD, System V, and DOS.

Introduction

Porting is important for a vendor in the open systems market. There are many reasons for this:

- Rapid advances in technology are creating new platforms at an astounding rate. It is essential that existing software be made available on new platforms as quickly as possible.
- Being able to port to existing customer equipment has clear financial and marketing benefits.
- Heterogeneous networks are becoming both larger and more common. The vendor's software product must run, and run well, in these environments.
- Large scale portability allows deploying the software on platforms with best price/performance.
- The reliability of the code is improved, as the differing environments provide different checks and constraints on the software.
- Widespread portability gives a leverage on testing. If the software works in some environments but not others, attention can be more quickly focused on the relevant areas. Additionally, different environments may have different testing tools. Exposing the problem on a platform with better testing tools can lead to more rapid repair.

All of the above are needed for both mature products and those currently undergoing large scale development. Responsiveness to market need is critical for competitive reasons in this environment. This means that the porting process must be fast, inexpensive, and robust. That the process should be fast and inexpensive should not need any further explanation or justification. In fact both these criteria will be sacrificed if necessary to ensure the process is ''robust''.

To explain what we mean by a ''robust'' porting process, assume that there is a software system called **Z** and that there are three platforms (*alpha*, *beta* and *gamma*) on which **Z** is to be supported. **Z** was ported to *alpha* in the past, but has not been or cannot be reconstructed or tested for sometime. **Z** is currently being maintained and tested on *beta*, and is under continuous development. **Z** has never been ported to *gamma*, but the sales department has told an important customer that **Z** already runs on it.

Our problem is then to port **Z** to *gamma*, quickly and completely, while ensuring that:

1) the modifications made to port **Z** to *gamma* do not adversely affect the ongoing

development on *beta*, yet can be easily and reliably integrated into the source once the port has been completed;

2) development done on *beta* (also promised by the salesman) can be added to *gamma* quickly and reliably without requiring any additional work beyond recompilation and test;
3) there is a high degree of confidence that the modifications made to port to *gamma* and enhance **Z** on *beta* will work on *alpha*.

In this paper, we will describe part of our porting strategy – how we specify the target system's characteristics – and explore some of the ramifications for the overall strategy, particularly with respect to achieving robustness as described above.

Why adherence to standards is not sufficient

The advent of software standards for open systems (POSIX, ANSI C) has improved the situation, but has far from solved it. The problem of developing and maintaining software on multiple platforms persists. There are many reasons for this persistence:

1) Standards compliance cannot be enforced and is frequently weak. If non-compliance is found, the software being ported is forced to adapt, not the other way around.
2) The standards themselves are often moving targets, and, despite the best of intentions, cannot be complete.
3) Any sizable system requires the specification of a large number of controls and settings that depend of factors far beyond the scope of any standard (e.g., **–O** vs. **–g**, the directory into which the system is to be installed).
4) Standards usually define a minimal system and we need to be able to use extra ''non-standard'' facilities offered by the platforms that might improve performance or security.
5) There are still a large number of potential clients using non-standard platforms, that we do not want to ignore.

Hence we believe that adherence to standards is not sufficient, thus, the problem of the specification, determination, use, and validation of the platform and configuration dependent variations between systems must be solved.

The next section discusses our environment, principles we believe to be important, examples of specifications that must be handled, some constraints on a solution, and the basics of our approach.

Some of this work parallels the goals and objectives of Larry Wall's **metaconfig**, and Glenn Fowler's **#include <feature/*.h>** systems. However, our requirements and constraints are sufficiently different to require yet another solution, as will be discussed later in the paper.

Our environment

A brief description of our environment and the challenges to be faced:

We have approximately a dozen software engineers working on various libraries, daemons, and utilities comprising some 2000 C source files and 500,000 lines of code. The developers tend to do their development and testing on only one or two of the available platforms.

Over the past three years, there has been an average of 50 files changed a day.

These changes have been made and tested on all of the internal platforms averaging six different platforms, and nine different configurations.

In the past year, the software has been moved to ten new environments (six in the last four months). Some of these ports have been on very short notice.

Finally, most development is done on platforms that are not (officially) supported in the released product.

Assuring functional consistency and robustness is a challenge. We require a consistent and comprehensive porting strategy that works well within this environment because we have to ''port'' and test fifty deltas a day to nine different configurations, while ensuring that the changes will not break any of the other dozen or so supported platforms.

How we solved the problem

Principles

The following are fundamental principles of our porting strategy:

1) We do not port the software itself; instead, we configure a *platform base* or portability layer on which all code is based.
2) Testing must be frequent and widespread. If the portability layer is correctly configured and changes to product software uses that layer correctly, then a successful test of a change on a single platform **should** be sufficient to ensure that the change will be semantically correct on all platforms. Obviously testing on only one platform is not sufficient in practise, therefore we test all nine standard configurations continuously.

To facilitate using these principles, we also mandate

- The use of exactly the same application source files for all platforms (the ''one true source''), which in turn means
- avoiding **#ifdef** in application code – especially those that deal with system dependencies. This applies to application header files as well; system dependent values should be inherited from the platform base.

Types of problems encountered

There are many different types of platform differences that give rise to porting difficulties. Some are listed below, with common examples:

Include file problems: Differences in location information (are the *open()* flags in *file.h*, *fcntl.h*, or even provided at all) incompatibilities amongst vendor headers (multiple, distinct definitions of *NULL*), ordering dependencies due to lack of idempotency, etc.

Different names for the same function: *strchr()* vs. *index()*, *bcopy()* vs. *memcpy()*, etc.

Standard libraries that aren't: For a program that uses terminal capabilities, do you need *–ltermcap*, *–lterminfo*, *–lcurses*, or some combination?

Functions that have different types on different platforms: *char* sprintf* (BSD) vs. *int sprintf* (Sys V).

Presence or absence of a capability: Is *lstat()* available? Does *rm* of a symbolic link delete the link or the file behind it?

Runtime environment: Is the user's login name in *$LOGNAME*, *$USER*, or even available?

Construction differences: Is *ranlib* available? Is *__STDC__* defined, and can you believe it if it is?

Bugs: If the *rename()* function exists, does it work, and how well does it work? [1]

Differing tool interfaces and semantics: Is the debugging flag for *cc*, *–g* or *–gx*? Is the *ar –o* flag supported?

Attributes of a Viable Solution

There are several attributes that a solution within this problem domain must possess:

Extensibility: It has been our experience that every new port introduces a new variation that has not been seen in any of the previous ports. To preserve portedness of old systems over time despite changes we must therefore extend capabilities without breaking old ports.

Recreatable: We must preserve port information for old systems over time so that we can recreate that port or extend as required as new requirements arise.

Locale independence: The mechanism must be host, site, and user independent. For example, we construct our DOS version on a UNIX system. This implies that no aspect of any specific host is needed to maintain this mechanism.

Ease of use: Given the rapidity with which new ports must be done, it is necessary to have a mechanism that allows easy addition or corrections of these parameters. This also encourages ready experimentation.

The extensibility criteria dictates that this cannot be a fully automated process, with all relevant information determined at compile or run time. Additionally, some values cannot be determined automatically, as they are almost a matter of taste and local custom. Such preferences should be specified with the same mechanisms as the platform constraints.

Configuring the Portability Layer

The foundation of the portability layer is a set of configured header and data files. The portability layer also contains a compatibility library, run time configuration tools and techniques and data, and a highly configurable construction system used to perform system constructions, but all these components are built using the foundation.

The portability layer foundation is built from the following ingredients:

1) A parameters file for the target system to be constructed.
2) A set of prototypes for files to be configured.
3) A program called *strfix* which, for a given prototype and the parameters file produces the configured information.

Descriptions of each of these ingredients follows:

The parameters file: The parameters file contains the configuration or discrepancy specifications as *name/value* pairs. The name is a upper case C identifier, and the value is an arbitrary string. Inclusion of other parameter files is supported to allow the inheritance of common or base systems values, as well as other shared information. An annotated example is provided in a later section.
Multiple specifications for a given name are allowed, with the last specification taking effect.

Prototype files: A prototype is a standard text file with embedded strings of the following forms:

```
@<name>@
@<name><operator><argument>@
```

where `<name>` is a possible parameters file setting, and `<operator>` is used to indicate special interpretation such as ":default" to specify a default value.

strfix: The program **strfix** reads a prototype file, replacing embedded **@<name>...@** strings with corresponding values from the parameters file. All other text is passed through unchanged.

The process

Creating the portability layer foundation is simply a matter of applying **strfix** against all the prototype files using the parameters file to produce a set of configured files which are copied to their proper locations.

Validation

Validating the portability layer is done, in part, by using it to compile and install the portability layer tools, which are then applied to rebuild itself. However, this is not an exhaustive test, consequently there are a number of regression tests that attempt to provide complete coverage.

This process is obviously dependent on the correct functioning of the **strfix** program. To minimize the chances that a new platform will force a change to **strfix**, its functioning has been kept very simple and easily tested.

Requirements of the construction system to support this strategy

This strategy has obvious implications for both the use of header files and the characteristics of the construction system. These will be addressed in a later section.

Prevention of gratuitous timestamp propagation

Obviously, every file in the portability layer depends upon the parameter file. Just as obviously, lots of programs depend on the portability layer. Therefore, unless other steps are taken, a cosmetic change in the parameters file would result in the complete and gratuitous reconstruction of the entire system.

Aids in preparing the parameters file

Whereas other approaches try to probe the system to determine the settings of various system values (and then automatically use these values), we do very little interpretation of the host environment, other than a program to extract required manifests from *sys/param.h* — a file we are anxious to avoid.

Such probe programs are particularly vulnerable to unanticipated values (or new parameters) forcing a coding change in the probe program, which then makes it difficult to assure that the new program would work correctly on previously ported systems.

Nearly all of our values depend on the parameters file. Instead of probing, we have tools to help the user to prepare and correct the parameters file.

Construction system implications

We are highly dependent upon having a construction system that will guarantee that a construction rule will be automatically applied whenever it is necessary – dependencies are automatically tracked and a changed dependency list or recipe forces reapplication.

The second requirement is that we adopt a programming style that uses our generated header files in lieu of the system provided header files.

If a discrepancy arises in a standard host header file, this ensures that we do not have to change the host header file. Hence we provide header file wrappers for all system header files that are used in the application code.

Note that this provides the necessary insulation from system dependencies that allows the programmers to ignore the underlying header structure. They need only use the generated header files rather than the system headers directly.

Annotated Examples

To describe in full the parameterization system would require the inclusion of a lot of documentation. Therefore, the following brief annotated examples are presented to clarify some of the issues presented in this paper[1].

The following is the parameters file for the optimized, X11R4 BSD4.3 side configuration of our product.

```
# SID @(#)mips4.5b-nix 1.18 ...
include         DefaultConf
include         Sites/snitor
include         Platforms/mips_4.3b
CONFIG          mips
C_OPT           -O -systype bsd43
HOSTNAME        helium
OPTIONS         DTMACH NO_MAN
RDIST           # nixtdc:/u/dtree/mips
XSYS_VERSIONX11R4
```

The *include* lines act as one would expect. Note that the last setting specified is the one that takes effect, therefore the setting for OPTIONS will override those specified in the default specification file *DefaultConf*.

Also note the presence of an SCCS SID line. This file is source and is subject to all the normal source controls. This file is the only specification used to parameterize the construction and this file is the only one that will differ from source used to build any other configuration. To build and install the specified configuration the only required initial human action is to specify this file to the initial configuration setup command. Once that specification has been stated, the file is an inherent part of the source and is subject to the same dependency tracking and rules as any other source file. A change to the file itself or any of its component files will result in the rerunning of any of the construction processes that use it as an input.

The following lines are a subset of the *TreeConfig* prototype file which itself is a *strfix* input file used to configure the construction process.

```
# SID @(#)TreeConfig.D 1.11 ...
# This file configured from @__FILE__@
OPTIONS @OPTIONS@
@{ @BLIT:false@
```

[1]Some example lines are truncated to ensure that they fit within the two column format requested by the programme committee.

```
@| true
       BLIT true
@| false
@| *
   @! BLIT(@BLIT@) must be true, ...
@}
XSYS_VERSION @XSYS_VERSION@
```

When this file is processed by *strfix,* given the example parameters file, the following will be output:

```
# SID @(#)TreeConfig.D 1.11 ...
# This file configured from /n3/...
OPTIONS DTMACH NO_MAN
XSYS_VERSION X11R4
```

The *@OPTIONS@* and *@XSYS_VERSION@* in the original strings in the prototype file have been replaced by the settings specified in the parms file. The `@{` through `@}` lines are a case statement based in the value of `@BLIT:false@`. If the *BLIT* parameter is `true`, `false` or unspecified (defaults to `false`) the lines immediately following the `@| true` or `@| false` line are processed up to the next `@|` or `@}` line. The `@|` arguments are one or more shell-like regular expressions, hence `*` will match everything, thereby processing any value that is not `true`, `false`, or unspecified. The `@!` string causes *strfix* to abort with a diagnostic thereby providing a quick and fool-proof check that the parameter is one of the legitimate values.

This is not an untypical example of a prototype file. Many are not C source and many are themselves used to configure other aspects of the system. The BLIT parameter is also a good illustration of a fundamental principle of our approach. It is used is one and only one place in the prototype files and it need not appear in any parameters file other than one for a system that indeed supports a BLIT. Thus no previous configuration file needs to be changed. Furthermore, the addition of this parameter will not change any existing generated file unless the BLIT parameter's value is true, thereby ensuring that no previous port is broken.

A typical parameters file, when the includes are unfolded, contains the settings for about 130 parameters. The number varies according to the target system as only variations for the default values are usually specified in the parameters files themselves and new parameters are created during most ports. This sounds intimidating, but for the most part just including the base system file (e.g., bsd4.[123], unix5.[0-4]) is sufficient to get started. The validation process quickly finds inappropriate settings that are fairly easily fixed by adding the appropriate override to the platform file. To list all the parameters is beyond the scope of this paper. However, they can be roughly partioned into the following groups:

Site information: the site addresses and phone numbers used within various packages to build new source files;

System configuration: the name of the system, the flags to be used to compile it (e.g., **–O** vs. **–g**); the target location for the installed product; the name to be used to access the installed product (frequently not the same as the target location[2]).

Header file mappings: the header file used to retrieve various types, settings, defines, etc.;

Tool names and availability: the name of the compiler, loader, *yacc,* etc. to be used in the construction process, as well as the names (and full path if desirable) of various useful facilities whose names may differ across systems (e.g., *rsh* vs. *rcmd*);

Compiler characteristics: Can one use prototypes? Can one use prototypes for function pointers? Some compilers can do one but dump core when one attempts the other. Is *char* signed or unsigned? How does one specify the use of an alternative C preprocessor? Tools are provided to help the installer find out answers to some of these questions, but we never depend on them working correctly.

Routine mappings: which routine should be used to copy memory? To move memory? Is there a *dup2* routine?

Supported bugs: Does *rename* work? and so on.

Before leaving this examples section, our solution to a particularly difficult header file mapping is illustrated.

The specification of which header files contain the definitions of the *tm* and the *timeval* structs is one which cannot be based on loose specification of the base system (e.g., bsd vs. unix5). The two structs are frequently used in the same source and the inclusion of the appropriate header files (e.g., *sys/time.h* and *time.h*) is not a matter of simply *#include*ing both. In some situations one includes the other and the second is not idempotent (that is, it may not be included twice). On other systems both files have to be included in a specific order and may or may not require the previous inclusion of *sys/types.h,* which itself is sometimes not idempotent.

We solve this problem by creating our own *envir/time.h* header file which contains the specification of both required structs, plus those prototypes that are sometimes not specified. This is a configured file that uses two parameters: **TIMEVAL_H** and **TM_STRUCT_H** which respectively name the header files to be used to access the definitions for the *timeval* and *tm* structs

[2]We test systems before we install them.

respectively, with one minor caveat. If one header file enforces the inclusion of the other, the other is assigned the empty string. The following lines are included as part of our prototype *time.h* file:

```
/*
 * include our own idempotent
 * types.h wrapper
 */
#include<envir/types.h>
/*
 * include timeval header
 * if necessary
 */
@TIMEVAL_H#i@
/*
 * header containing struct tm
 * if not same as above
 */
@{ X@TM_STRUCT_H:sys/time.h@X
@| X@TIMEVAL_H@X
@| *
   @TM_STRUCT_H#isys/time.h@
@}
```

The `@<name>#i...@` string causes *strfix* to output a `#include` line if the parameter is defined or a default value is specified. The above may look baroque, but the resulting file just consists of the comments and the required includes. Also note the absence of *#ifdef*s.

Evaluation.

How does this strategy meet the criteria given earlier?

- The simple text file is portable, and **strfix** is simple and almost immune to environmental idiosyncrasies.
- We can add new parameter files that allow us to use old configurations with new perturbations. For example, a quick look at the machine (or its documentation) will indicate whether we should start with a BSD 4.x base, or System V, or something else.
- Usually no application C code or header file changes are required. We will not discuss creating of a portability library to compensate for system deficiencies; it is a simple application of the parameter file to select or provide appropriate functionality or provide name mapping.
- In the last year, we have ported our major software product to ten new platforms. Six of these were in the last four months, for an average of one port every three weeks. These ports included our first encounters with X11R4 and System V Release 4. The porting itself took an average of a day, with testing taking a week. During these porting efforts, development efforts continued at their normal rate on the application code.
- Through this mechanism we have virtually eliminated the use of **#ifdefs** in C code. During the last year, with its ten ports, there were no **#ifdefs** added anywhere in the application code or header file. Those **#ifdefs** that remain are either in taste or capability selection (e.g., build with debugging code in or out) or are based on settings in the parameter file. When it is required to alter a setting for a specific platform or host, the parameter file is changed, and not the C code. The importance of this to our efforts should be obvious:
  + Since we do not change the code, we virtually eliminate the possibility of breaking an old port.
  + By not creating platform-specific blocks of code, the regression tests remain accurate.
- The ability to use these techniques in a cross-compilation environment allowed porting the code to a DOS/Windows environment without forcing an unfamiliar development environment on the developers. A probe-based mechanism would not have readily permitted this. As importantly, it permitted application of multiple developers to the porting effort without jeopardizing source code consistency. Although this effort did in fact require substantial code changes due to the radical environmental differences between DOS and UNIX (filename syntax, environment variables, unusual C environment), these changes were applied to the one true source for both the DOS and UNIX environments, and then continually tested in both environments on an ongoing basis.

How well does this work?

Very well indeed. This approach has succeeded in all UNIX platforms tried to date (over fifty at last count). Our approach is now being used to support our application code in a DOS/Windows environment.

Products at other Siemens sites have adopted this approach by converting their software to use our portability layer. One such product, which had previously only worked on one platform was ported to three new platforms in two months.

The application developers are, in our experience, very happy to suffer the style and coding practises in exchange for not having to understand the arcane topology of all the systems[3] encountered. The effort required of the developers to use the

[3]This became *very* important during the DOS/Windows work. Running the regression tests under DOS/Windows was all the contact most of the developers had with the DOS environment.

portability layer is very small by comparison.

Most ports of our software (up to and including running of the automated regression test suite) do indeed take only a day or so. The exceptions occur when the applications have made non-portable assumptions (e.g., using X11R3 and porting to a platform with X11R4). Even in these cases, the ability to rapidly mutate the platform base layer to adapt to the new environment without invalidating previous ports[4] is of great benefit.

It is worth noting another large benefit – anything that is a text file can make use of the platform base. This obviously applies to applications written in languages other than C, but it also applies to shell scripts, construction system recipes, application data files (e.g., X resource files), etc.

It is worthwhile casting our recent experiences into the **Z** software mold mentioned previously, with past platforms *alpha* which may not be testable for a while, present platforms *beta* on which most ongoing development occurs, and future platforms *gamma*. Platforms that were *gamma* systems have become *alpha* systems since the equipment was here for short term evaluation only; some of the future *gamma* systems[5] will be new corporate platforms, and will become *beta* platforms. Some of the *alpha* systems are now *beta* systems, as equipment has been repaired or returned. During all this activity, software development continued at close to its normal rate.

In such an environment of flux, it is clear that we cannot afford to freeze application development, port the code by modifying it and then test it on the relevant platforms. Attempting to modify the application for porting purposes while letting application development proceed has obvious quality problems. We are convinced that the more usual approaches would not suffice in our environment.

What would be done differently?

- Documentation of an individual parameter and the expression of its use is currently weak. This is especially important since each new port (so far) has introduced new parameters.
- Comprehensive validation of a parameter's setting is sometimes delayed until late in a system's construction due to prerequisites. We need a better framework for specifying and executing parameter regression tests.
- Similarly there needs to be an easy to use framework for adding aids to help the user determine the correct settings, although most of the time a simple guess is sufficient.

[4]In this case, the changes have to be tested both in an old (X11R3) and new (X11R4) environments to be considered safe.

[5]We have three of these anticipated in the next six weeks.

Other Approaches

Config

This paper would be incomplete without discussing a widely used approach to the problem addressed by this paper, namely Larry Wall's *config* and *metaconfig* systems. *config* is sufficient for the distribution of a small shareware system to users who are willing to invest the required time and effort to fix it when it goes wrong. However, *config* cannot be considered as the mechanism to be used to do large professional systems due to a number of deficiencies.

- It requires user interaction, which is time-consuming and error-prone, and most importantly cannot be expressed as an administered source file, something that we believe to be essential. It also rules out the possibility of rerunning the configuration stage as part of any construction, again something that we believe is essential.
- The use of probes to determine the appropriate settings for the equivalent of our parameters has several drawbacks. When a probe is in error, the probe mechanism itself must be altered to accomdate the fix. Frequently the probes themselves are constructed with implicit assumptions about the target system. When these assumptions are incorrect, major surgery is often required. Hence, previously valid probing assumptions may be upset by the change to the mechanism, jeopardizing the validity of previous ports.
- The probe information cannot be managed historically. The probe evaluation depended on the state of the host system at the time *config* was executed and therefore its replication cannot be guaranteed. Regenerating a configuration for an unavailable platform (say, an older release of an operating system) for support purposes becomes problematic.
- The addition of a new parameter or correction of an old one has severe performance implications when constructing large systems. Our experience with *config* is limited[6], but *config* users who do use it stated that the actual products of the *config* process are two configured files (one for C and the other for sh). This means that the simple correction of a parameter will require the entire recompilation of the complete system, something that one cannot afford when running four to five thousand compiles across nine different platforms.

[6]One of the authors tried to use it to install *rn* but gave up when it failed. The cost of trying to adapt a much hacked 1800 line shell script was considered to be far more than the benefits of being able to use *rn*. For comparison, we normally use a ten line text file to install 800 programs and 30 libraries.

- The dependence on the host system eliminates the possibility of doing cross compilation, something that we must have to adequately deal with inadequate systems such as DOS. Our approach is to provide probes that may be used to determine and/or test the appropriate value for a parameter, but to never incorporate its running as part of the construction process.

Fowler's #feature mechanism

Glenn Fowler, the creator of the fourth make, has an approach to the configuration process that to our knowledge and that of one of his colleagues has not been documented. Briefly, the use of:

```
#include <feature/name.h>
```

within a C program, and the dynamic dependency tracking of *make4*, will trigger, if necessary, the creation of the named header file by running the associated probe. This shares some of *config*'s weaknesses with respect to the dependence on automated probes and the host environment, but avoids some of *config*'s major flaws. As stated the system is, as yet, undocumented but shows promise.

Conclusions

Porting is extremely important to us, and our techniques have proven to be profitable for us.

This paper addresses only one aspect of the porting problem – that of the specification of parameters for a system. We have been led to this strategy by the requirements of today's environment of open systems and need of rapid ports. The parameterization and characterizations of systems in this way has proven sufficient to handle all porting problems we have seen in the past ten years. Indeed, expectations are now so high we have the situation that all ports are expected to be done in a day, even though they may involve substantial rework and testing to deal with new challenges.

Bibliography

[1] David Tilbrook, *rename(''open'', ''swinging_to_and_fro'');* EurOpen Newsletter, 1991.

[2] David Tilbrook & John McMullen, *Washing Behind Your Ears: Principles of Software Hygiene, EurOpen Nice Conference, Oct. 1990.*

Author Information

By the time this paper is published, David Tilbrook will have started his new position as Vice President, Technology at CS Computing Services in Toronto. For the last three years he has been a consulting engineer at Sietec and the manager of the Software Hygiene Research group. His primary research interest is Software Hygiene and the Software Process. David has served as the programme chair for four EurOpen conferences, a Usenix conference and the Software Management Workshop, and is the chair for the 1993 Uniforum Canada conference on Software Hygiene. In 1985, David was awarded an Honourary Lifetime membership to EurOpen (and a much treasured Swiss army knife).

David's new address is unknown at this time due to the fact that his company is relocating to as yet at unknown location in downtown Toronto but he may be contacted via Russell or `dt@sni.ca` for the time being.

Russell Crook has worked at Sietec since 1988 as a Project Leader within the Imaging and Data Storage groups, working on problems in large scale data storage and management. He has a long standing interest in computer chess, including work on the Treefrog chess program, which won the 1974 CACM computer chess tournament.

Russell may be reached at:

Sietec Open Systems Division
2235 Sheppard Avenue East
Suite 1800
Willowdale, Ontario
Canada
M2J 5B5

or at `rmc@sni.ca`.

Performance of a Parallel Network Backup Manager

James da Silva, Ólafur Guðmundsson, Daniel Mossé
– Department of Computer Science, University of Maryland

ABSTRACT

The advent of inexpensive multi-gigabyte tape drives has made possible the completely automated backup of many dozens of networked workstations to a single tape. One problem that arises with this scheme is that many computers cannot backup their disks over the network at more than a fraction of the tape's rated speed. Thus, running overnight backups sequentially can take well into the next day.

We have developed a parallel backup manager named *Amanda* that solves this problem by running a number of backups in parallel to a holding disk, then using a multi-buffer copy scheme to transfer the backups to the tape at the full rated tape speed. Amanda uses accurate estimates of current backup sizes as well as historical information about backup rates so as to schedule backups in parallel without swamping the network or overrunning the holding disk or tape.

Locally, we use Amanda to back up 11.5 gigabytes of data in over 230 filesystems on more than 100 workstations, using a single 2 gigabyte 8mm tape drive, taking two to three hours each night. This paper discusses the architecture and performance of Amanda.

Background[1]

Until a few years ago, the backup medium of choice for most large UNIX sites was the 9 track reel-to-reel tape, while 1/4" cartridge tapes were (and still are) popular with smaller systems. Storage capacities for 9-track and cartridge tapes vary from about 40 to 200 Megabytes. These tape systems are often of smaller capacity than the disk subsystems they are backing up, requiring an operator to feed multiple tapes into the drive for a full backup of the disks.

This problem has had a big influence on site system administration. Sites with only a few large timesharing systems or file servers can arrange backups by operators at scheduled times, but the coordination of backups of a large number of workstations on a network is more difficult. Requiring users to do their own backups to cartridge tapes doesn't work very well; even computer-literate users just don't do backups on a regular basis.

A common solution that most sites have adopted is a *dataless* workstation model, in which all user data is stored on file servers with small local disks to hold temporary files and frequently used binaries, or even a *diskless* workstation model, where the workstations have no disks at all[1]. These network organizations require fast file servers with large disks, and generate heavy network traffic.

Our department, on the other hand, has always used *datafull* workstations, where all user data, temporary files and some binaries, are stored on the workstations. File servers only provide shared binaries. This allows the use of smaller file servers, with smaller disks. A big advantage of this model is political; users tend to want their own disks with their own data on their own desks. They don't want to deal with a central authority for space or CPU cycles, or be at the whim of some file server in the basement.

Since most file writes are local, performance can be better as we avoid the expensive synchronous NFS file writes and network traffic is lower. With the datafull model we are able to have each fileserver support over 40 machines if needed, while in dataless and diskless environments only specialized fileservers can support more than 20 workstations. The big disadvantage is the difficulty of managing and backing up all the datafull workstations.

[1]This work is supported in part by contract DASG60-87-C-0066 from the U. S. Army Strategic Defense Command to the Department of Computer Science and by Rome Labs and the Defense Advanced Research Projects Agency under contract F30602-90-C-0010 to the UMIACS, both at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Department of Defense, DARPA, RL, or the U.S. Government, unless so designated by other official documentation. Computer facilities were provided in part by NSF grant CCR-8811954.

The arrival of inexpensive gigabyte Digital Audio Tape (DAT) and 8mm tape technology has changed the situation drastically. Affordable disks are now *smaller* than affordable tape drives, allowing the backup of many disks onto a single gigabyte tape. It is now possible to back up all the workstation disks at a site over the network onto a single 8mm tape.

Now that the space problem is solved, the new problem is *time*. Backing up workstations one at a time over the network to tape is simply *too slow*. Many workstations cannot produce dump data as quickly as tapes can write[2]. For example, typical dump rates (both full and incremental) on our network range between about 5% to 70% of the rated 246 KB per second of our Exabyte EXB-8200 8mm tape drives[3]. We found that we could not add workstations to our network backups because the nightly backup would not finish until well after the start of the next work day.

Amanda, the ''Advanced Maryland Automated Network Disk Archiver,'' was developed to solve these problems. To make the project manageable, we built Amanda on top of the standard BSD UNIX DUMP program. Amanda uses a holding disk to run multiple backups in parallel, and copies the dump images from the holding disk to tape, usually as fast as the tape can stream.

This paper concentrates on the performance issues involved with backing up a network of datafull workstations, including the performance characteristics of DUMP, tape drives, and of the sequential and parallel versions of our backup manager. We will also discuss the architecture and technical details of Amanda.

Performance of BSD DUMP

Berkeley UNIX systems and their derivatives come with a backup program called DUMP, and its corresponding restoration program, aptly named RESTORE. DUMP can do incremental and full backups on a per-filesystem basis. Incremental backups are done in terms of numbered *dump levels*, where each level backs up all the files changed since the last backup at a lower dump level. Full backups are known as *level 0 dumps*. While backup policies vary from site to site, generally a level 0 dump is done on each filesystem once every week or month, and incrementals are done every day. We run incremental backups every weekday evening, and each filesystem gets a full dump every two weeks.

RESTORE can restore entire filesystems or individual files and directories, and includes an interactive mode where operators can browse the dump directories to pick what should be restored.

DUMP runs several child processes that read particular files from the disk in parallel and take turns writing to the output tape. In this way DUMP is able to keep the tape writing and disk reading simultaneously. Once DUMP has decided which files to back up, it produces data at a very steady rate for the duration of the backup. We have measured the dump rates over the network for various computer architectures; some typical values for full dumps are shown in Table 1. These numbers will depend on disk speeds and compression rates as well as architecture, so your mileage will vary.

To get more data onto backup tapes, it is often advantageous to compress the backup data. Table 1 also shows the resulting dump rates when the dump output is run through the UNIX COMPRESS program before being sent over the network. The effective dump rate column is the *original* dump size divided by the time it took to dump with compression.

Architecture	Dump Rate	Compressed Rate	
		Actual	Effective
SPARCstation 2	322	90	150
SPARCstation 1+	172	45	101
DECstation 3100	142	38	101
NeXT Cube	164	18	36
VAXstation 3200	128	15	40
Sun 3/50	124	12	29
Sun 2/120	32	4	7

Table 1: Full Dump Rates (KB/sec)

These times are for level 0 dumps on relatively large disks. Incremental dumps will have much lower rates, because there is a fixed overhead for DUMP to scan the filesystem before dumping. Incrementals have less data to amortize that overhead, as shown in Table 2.

Architecture	Dump Rate	Compressed Rate	
		Actual	Effective
SPARCstation 2	298	60	156
SPARCstation 1+	22	10	19
DECstation 3100	47	11	40
NeXT Cube	39	11	20
VAXstation 3200	3	1	3
Sun 3/50	53	8	24
Sun 2/120	15	4	4

Table 2: Incremental Dump Rates (KB/sec)

With each filesystem getting a full backup only once every two weeks, each night about nine out of ten filesystems are getting an incremental dump, making the incremental backup rates more representative of the overall backup rates.

Before DUMP outputs any data, it makes an estimate of how large the output will be. This estimate can be used to make calculations about dump sizes in advance. We measured the accuracy of this estimate as a predictor of dump output sizes for dumps done later the same night. The results are good: the dump estimates are very accurate, usually well within 1%.

Like most backup systems, DUMP does have some problems correctly backing up filesystems that are being modified while the backup is occuring[4]. Some sites instead use file-oriented programs, like TAR or CPIO, but these programs have their own set of problems[5].

Performance of Gigabyte Tape Drives

The two competing gigabyte-class tape technologies are the 8mm Exabyte Cartridge Tape drives and the 4mm DAT drives.

The Exabyte EXB-8200 [3] is rated to hold up to 2500 megabytes of data, and stream at 246 KB/s. According to our own measurements, we get 2200 MB on our tapes, and a transfer rate of 238 KB/s. We have measured the filemark size to be 2130 KB, and it takes about 10 seconds to write.

Many vendors sell DAT tape drives. We have on hand a DEC TLZ04 Cassette Tape Drive[6]. We do not run Amanda on this drive, but we measured some of its characteristics for comparison. The drive is rated to hold 1.2 gigabytes, and transfer data at 183 KB/s. According to our measurements, we get 1245 MB on our tapes, and a transfer rate of 173 KB/s.

Achieving Rated Tape Speed

We measured the transfer rates to tape by repeatedly writing a memory buffer on a system with no load, that is, writing in a tight loop as fast as the operating system (OS) and tape would allow. Achieving that same rate when transferring dump image files from the disk to the tape requires some sort of multi-buffer technique to keep the tape writing and the disk reading at the same time. There are several ways to do this.

Using just traditional UNIX pipes for synchronization, multiple processes can do the I/O, where each process reads from the disk and takes turns writing to the tape (this is the approach used by DUMP). If the OS allows shared memory, two processes, a reader and a writer, can share a pool of buffers. Or, if the OS supports asynchronous input/output, both reads and writes can be outstanding with a single process.

We have implemented these techniques and find that, when properly tuned, they can achieve the same rate transferring large disk files as when writing a memory buffer in a tight loop. The number of reader processes or memory buffers needed is system dependent, and is more than might be expected in theory. The sibling processes can produce or consume many buffers before the others get a time-slice,

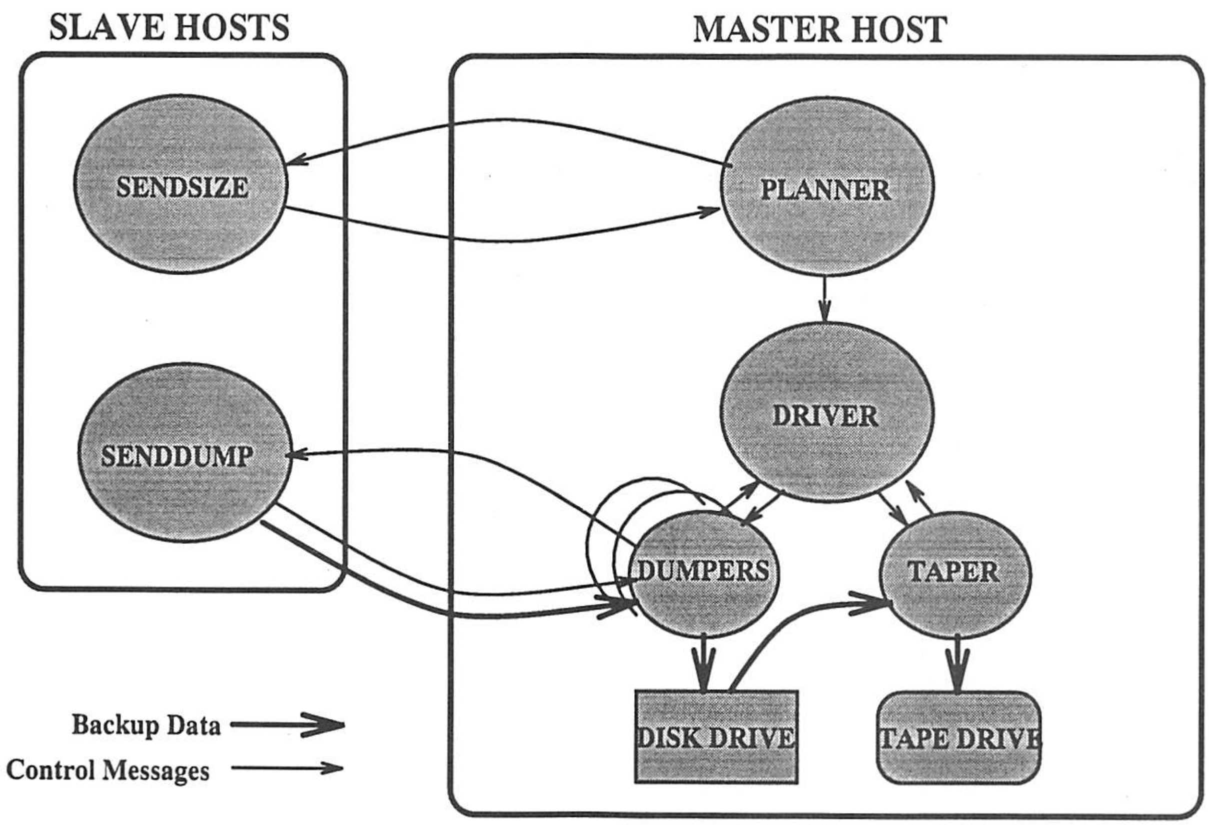


Figure 1 – Architectural Components of Amanda

because of coarse scheduling granularity, read-ahead on the disk, and buffering in the tape drive.

Small files never reach the rated speed because there is not enough data in the files to cause the tape drive to stream efficiently. The time to write small files is dominated by the filemark write time.

Amanda

Amanda is a batch oriented backup manager, invoked each night through CRON on the *master host* to back up a list of filesystems from across the network on *slave hosts* to a multi-gigabyte tape drive.

The original version of Amanda executed remote dumps to the tape drive one after another according to a dump list. It took care of handling failed and timed out dumps, mailing a status and error report to the operators, and preparing the next night's dump list. We ran this system for over a year at three sites here at the University of Maryland.

While we liked the reporting, error handling, and schedule management features of our backup manager, it was too slow to handle all our workstations in a single night's run. It was taking consistently seven or eight hours to back up only 600-900 MB each night. Occasionally something would go wrong and dumps would run until noon the next day.

We solved this problem by completely redesigning the system to run enough dumps in parallel so that the cumulative dump rate matches the tape speed. We can run the dumps in parallel while writing them to tape sequentially if we use a disk as a buffer to hold the dump data while the parallel dumps are running. This disk is called the *holding disk.* Once a dump is finished, the dump image is copied from the holding disk to the tape as fast as possible, then deleted. Any dumps that are too big for the holding disk are dumped sequentially to tape after all other filesystems have been dumped. When there is a tape problem, Amanda will run incrementals of all filesystems to the holding disk, which can then be flushed to tape when the operators arrive the next morning and correct the problem.

The filesystems can also be compressed before being transferred, which results in about a 40%-60% reduction in output size. When dumping several filesystems in parallel, compression does not adversely affect the total Amanda run time. In fact, the reduction in output size reduces the total time because there is less data to write onto tape.

The components and data flow of Amanda are shown in Figure 1. Amanda runs in two distinct phases. The first phase, called PLANNER, manages the overall backup schedule. It is responsible for keeping the schedule balanced and for assigning dump levels for each filesystem for the current Amanda run. PLANNER outputs its dump list to the second phase, called DRIVER. DRIVER executes all the dumps, deciding what order to run the dumps, and what order to write them to tape.

PLANNER uses accurate estimates of backup sizes along with historical data on previous dumps (including backup rates and compression ratios, to assign a dump level to each filesystem). If the set of backups to be done is too large for the tape, some full dumps are delayed for one night. If current backups are too small relative to other nights in the backup cycle, some full backups originally scheduled for the following night are moved forward. In this way the backup schedule expands, contracts, and balances out automatically as needed.

```
From:     bin@cs.UMD.EDU
Subject: CSD AMANDA MAIL REPORT FOR March 28, 1992
To:       csd-amanda@cs.UMD.EDU

These dumps were to tape VOL15.
Tonight's dumps should go onto tape VOL1 or a new tape.

STATISTICS:
                               Total      Full      Daily
                             --------   --------   --------
Dump Time (hrs:min)             2:54       1:25       1:15   (0:14 taper idle)
Output Size (meg)             1438.1     1070.9      367.2
Original Size (meg)           2165.6     1476.4      689.2
Avg Compressed Size (%)         62.4       70.2       44.0
Filesystems Dumped               236         17        219
Avg Dump Rate (k/s)             40.1       51.8       24.1
Avg Tp Write Rate (k/s)        152.9      214.9       83.1
```

Figure 2: First Page of an Amanda Mail Report

PLANNER gets its accurate dump size estimates from the slave hosts themselves. A datagram is sent to the SENDSIZE service on every slave host in the network, requesting estimates for particular dump levels for each filesystem. SENDSIZE runs DUMP to get each needed estimate, killing dump once it outputs its estimated size. The estimates are then sent back in a datagram to PLANNER on the master host. Depending on the number of requests and the size of the filesystems, this procedure takes about 2 to 5 minutes per slave host. However, PLANNER sends its request datagrams to all the slave hosts first, then waits for replies to come in. Therefor, the entire PLANNER phase takes about 5 minutes, regardless of whether there are 20 slave hosts or 120.

For filesystems doing incremental dumps, estimates are requested both for the current level that the filesystem is dumping at, and the next higher level. If dumping at the higher level would produce a much smaller dump, and if the filesystem has been dumped at the current level for at least two days, the level is *bumped* by PLANNER. Automatic bumping provides a good tradeoff between incremental dump sizes and the desire to have fewer dump levels to restore. The user controls the bump threshold.

PLANNER outputs its decisions and estimates to the second phase, called DRIVER. Staying within user-specified constraints on total network bandwidth allowed, total holding disk space allowed, and maximum number of dumps in parallel allowed, DRIVER runs as many dumps as it can in parallel to the holding disk. As dumps to the holding disk complete, they are transferred to the tape by the TAPER program. TAPER consists of both a disk reader and a tape writer process, with shared memory buffers between them. The reader and writer parts of TAPER fill and drain buffers asynchronously.

DRIVER directs a number of DUMPER programs, one for each dump that can run in parallel. DUMPER connects to the SENDDUMP program on the slave host, receives the desired backup image and puts it on the holding disk. SENDDUMP parses the control message output of DUMP for any failures or error messages (like read errors on the slave's disk) and DUMPER logs these problems.

In addition to these main components, Amanda includes several other auxiliary programs. A report writer scans the logs of the latest Amanda run and sends a report to the operators (see Figure 2 for an example). Amanda tapes are labeled, and Amanda will not write to an unlabeled tape, or to a labeled tape that contains recent dumps that should not be overwritten. This prevents some common operator errors (such as forgetting to change the tape) from causing loss of data. Instead of failing completely, Amanda will run incrementals of all filesystems to the holding disk, which can then be flushed to the right tape when the operators arrive the next morning and correct the tape problem.

A Performance Experiment

While Amanda's backup management features are useful even without parallel dumping, it is the parallelism that gives a big win over other solutions. To determine the effect parallelism had on our backup times we conducted an experiment. Late on Saturday night of SuperBowl weekend, when the network and machines were unusually idle, we ran Amanda repeatedly with the maximum number of DUMPERs allowed varying from 1 to 11. We configured Amanda to run all the dumps through COMPRESS on the slave host.

PLANNER was run once to generate the list of filesystems to be dumped. For this experiment, 178 filesystems on 79 hosts (25 Sun3s, 31 SPARCstations, 23 DECstations) were dumped. The total size of the dumps, after compression, was 498 MB. There were 21 full backups accounting for 404 MB, and 157 incremental backups, taking 94 MB. Getting the estimates from all 79 slave hosts and making its decisions took PLANNER less than 5 minutes.

Our dump master host was a SPARCstation IPC with 16 Megabytes of memory, an Exabyte EXB-8200 tape drive, and a Fujitsu M2266 1.2 gigabyte disk drive, with Amanda configured to use a maximum of 800 MB for its holding disk. In this experiment, only 403 MB were actually used.

Figure 3 shows the total run time of the experiment as a function of the number of DUMPERs allowed. The first curve shows the total time to complete all the backups to tape. It drops drastically because the tape can write backup data at its full speed concurrently with several dumps that are running to the holding disk at slower speeds.

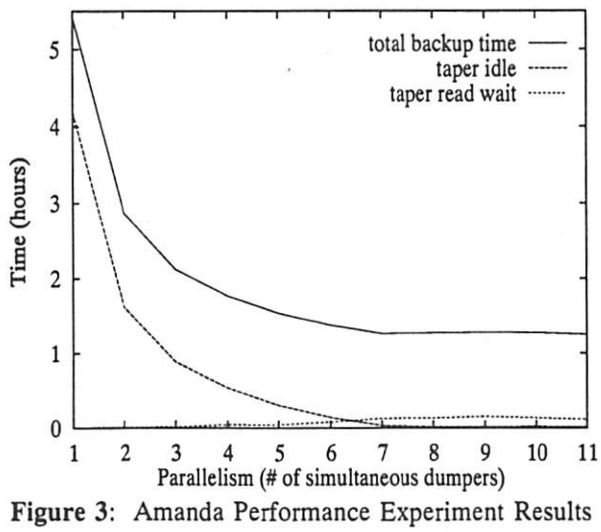


Figure 3: Amanda Performance Experiment Results

The second curve shows the amount of time TAPER is idle, waiting for dumps to finish before transferring them to tape. It is this idle time that is responsible for the relatively long run times at low levels of parallelism; there are not enough dumps finishing to keep TAPER busy. At 7 or more DUMPERs, TAPER is kept busy virtually 100% of the time, so the curve levels off.

Remember that TAPER consists of both a reader and writer process. The third curve in Figure 3 is the cumulative time the writer process had to wait for a buffer to be read into memory. When the multiple-buffer copy scheme is working, this time is close to zero — the writer always has a full buffer when it needs one. However, when the reader has to contend with many DUMPERs for access to the disk it can fall behind the writer, causing the TAPER to slow down, and the total dump time to increase slightly. As the load on the master host increases, we observe a general degradation of performance, caused by factors such as tape write time and control message transfer times which we did not measure directly.

Operational Experience

Running network backups in parallel has been a big win for us. What used to take us six or seven hours with sequential dumps now takes approximately 75 minutes. This has enabled us to add *all* of the rest of our workstation disks to the backup system, taking over for users that were previously supposed to be dumping their disks to cartridge tapes themselves. We are now backing up 11.5 gigabytes of data in over 230 filesystems on more than 100 workstations, in two to three hours.

We have found that each night incremental dumps account for roughly 1/3 of the total output size. We have also found that 500 MB of filemarks are written each night. Given these two factors and a measured tape size of 2200 MB, there is room for about 1100 MB of compressed full dumps each night. With ten nights in our backup cycle, and compressed sizes at about 60% of the original sizes, we can back up 18000 MB with our single Exabyte 8mm drive.

By those calculations, we still have room for about 6 more gigabytes of data, at which point we will be filling a 2 gigabyte tape in about 4 hours with Amanda, compared with about 16 hours dumping directly to tape. More data can accomodated after that point by lengthening the dump cycle to more than two weeks between level 0 dumps.

Simulating Amanda Performance

Our initial DRIVER algorithm to schedule dumps was very simple. It took the PLANNER output and executed dumps on a first-fit basis, running whichever would not overflow the holding disk, the total network bandwidth, or the number of DUMPERs. When a dump finished, it was put in a first in, first out (FIFO) queue for writing to tape.

This simple algorithm does very well most of the time, but occasionally fails to keep the tape busy, slowing down the backup procedure. For example, when DRIVER does not start a large dump soon enough, everything else will finish well before this dump, leaving TAPER with nothing to do until the dump finishes.

In order to evaluate the performance of our algorithms in a variety of environments, we implemented a trace-driven simulator testbed. By plugging proposed driver algorithms into our simulator we can easily measure the impact of any changes. The simulator uses performance results from our actual nightly runs. It is thus very accurate; it matches real results to within a minute or two, which represents about 1% error, due to various system overheads that the simulator doesn't account for.

We have evaluated several algorithms using the simulator and have settled on one that orders the list of dumps by time, and places DUMPERs on both ends of this sorted list. The distribution of dump times (see Figure 4) is the key to keeping the tape busy. Most dumps are small incrementals that finish very quickly. Since our tape takes ten seconds to write a filemark, the very smallest dumps can complete quicker than they can be written to tape. The first incremental to complete, usually from a quiet filesystem on a fast machine, is done in 2 or 3 seconds. By the time the tape can write the filemark for this dump, the next two tiny dumps will be finished and will be ready to be written. In fact, almost 25% of the dumps take less time than the filemark write time, so, one or two DUMPERs can keep the TAPER busy for all the smaller dumps.

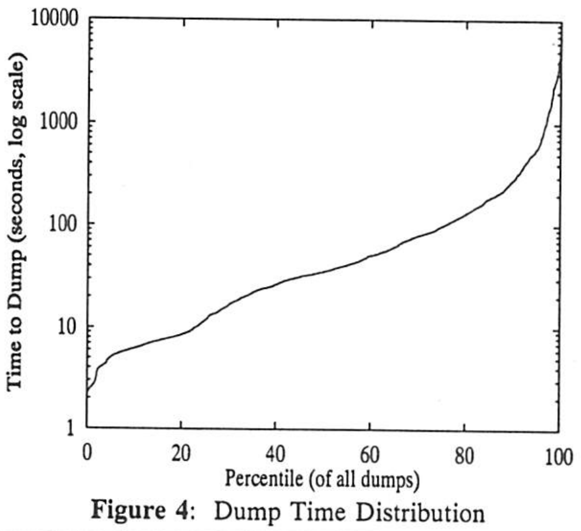


Figure 4: Dump Time Distribution

Less than 10% of the dumps take more than 5 minutes. But of these, some can take a very long time, particularly full backups of big disks on slow workstations. It only takes a handful of very long dumps to dominate the entire run time, so it is important to start the longest dumps as soon as possible. Since one or two DUMPERs can handle the majority of dumps starting at the short end, it makes sense to concentrate all the rest of the DUMPERs on the long end of the dump time distribution, making certain some of the longer dumps are started right away and thus finish before TAPER goes idle. This algorithm is based on a variation of the first-fit decreasing bin-packing algorithm[7].

We also studied variations for the queue of finished dumps waiting to be written to tape. As we expected, ordering the queue by *largest file first out* (LFFO) performs significantly better than our original FIFO queue. Writing the largest file first then deleting it reduces the load on the holding disk quickly, making room for new dumps to run.

Figure 5 compares the run time, according to our simulator, of traditional sequential dumping, our initial DRIVER algorithm, and our current algorithm, executed with actual dump data from 25 nights of dumps at our site. The parallel algorithms are not as sensitive to individual dump speeds as the sequential backup procedure. The parallel dumping of several file systems makes it possible to absorb the time for longer dumps while dumping several other smaller or faster machines.

The increase in simulated time for the sequential dumpes in the interval between 5 and 20 nights is due to the steady increase in the number of file systems being dumped, as we added disks to our Amanda backups. Note that the current algorithm shows very little fluctuation of the run time. The peak observed on the 23rd night is due to a tape

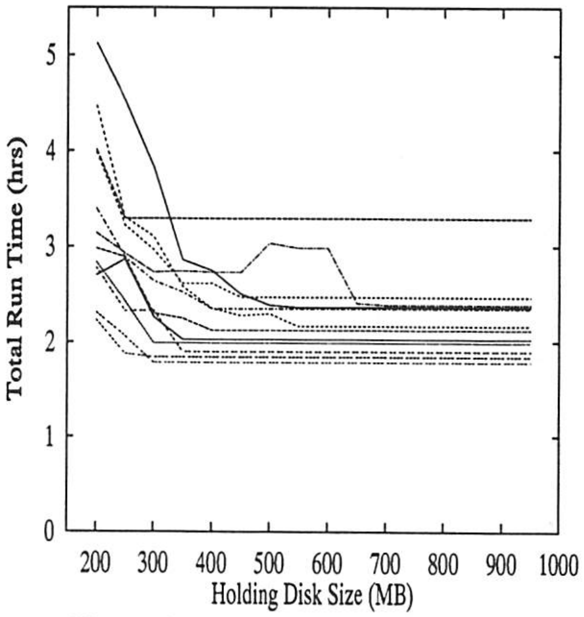


Figure 6: Effect of Holding Disk Size

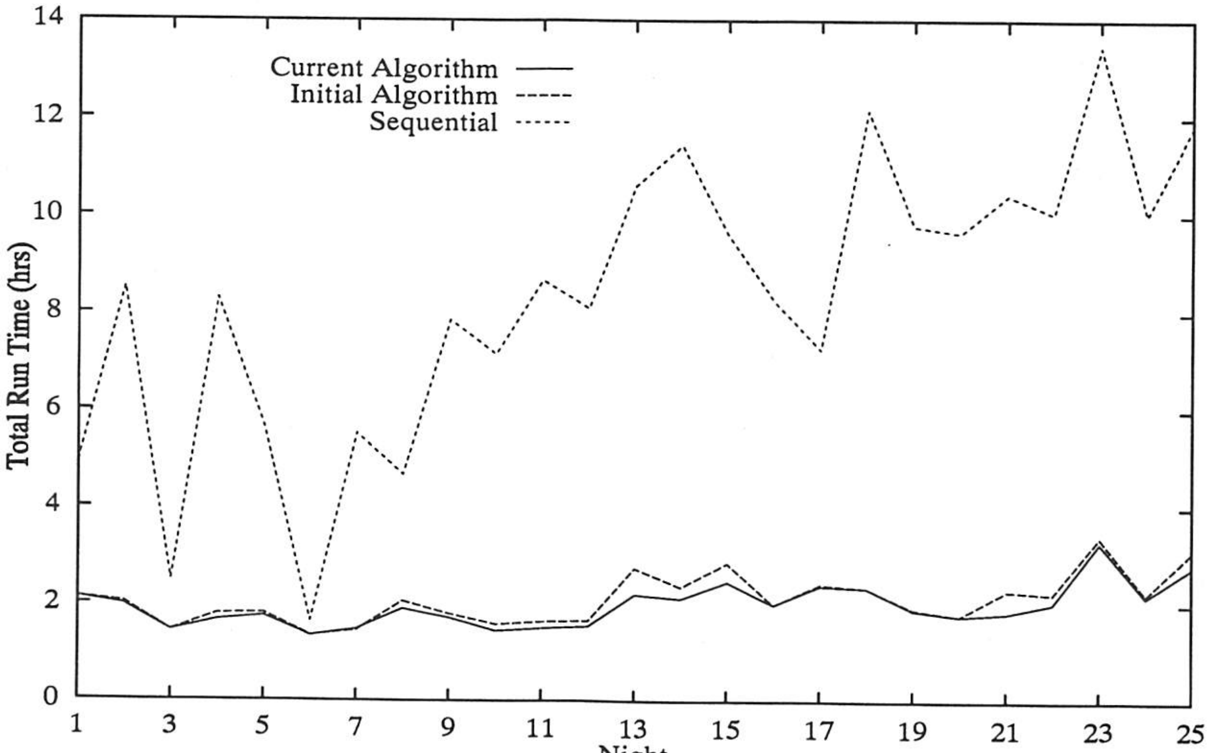


Figure 5: Run Times of Various DRIVER Algorithms

failure that occurred the night before (that night is not shown in Figure 5), and therefore the amount of data to dump on the 23rd almost doubled.

While we run with a large holding disk, Amanda can perform well with much smaller holding disks. Figure 6 shows the total dump times, according to the simulator, for 12 different nights with the holding disk size varying from 200 MB to 950 MB for each night. Even with holding disk sizes as small as 200 MB, Amanda performs much better than sequential dumps.

Conclusions and Future Work

With Amanda, we have achieved performance close to optimal, that is, dumping filesystems as fast as we can write files to tape. This was done with 7 or 8 DUMPERs. We have studied and continue to examine different scheduling algorithms for the DUMPERs, in order to minimize the time to complete backups, while remaining within the resource constraints. Our current bottleneck is the tape speed, but we believe our algorithm is sound enough to address different constraints in a wide variety of installations.

We are currently considering a prescheduler that determines the start time of every dump before any dumps are done. This scheme, if feasible, will further reduce the number of DUMPERs needed to keep the tape busy, and will minimize the amount of holding disk space needed, by spreading out the dumps so that they are ready just when TAPER needs them, and not before.

Our future work will focus on generalizing the scheduling algorithms to address constraints from incoming technologies, such as faster tape systems, larger disk subsystems, larger scale in terms of number and speed of machines, other backup subsystems such as CPIO and GNU TAR.

Regardless of any future improvement, we have already achieved our goal of backing up all our workstation filesystems to a single 8mm tape overnight with no operator intervention.

Software Availability

Amanda is copyrighted by the University of Maryland, but is freely distributable under terms similar to those of the MIT X11 or Berkeley BSD copyrights. The sources are available for anonymous ftp from **ftp.cs.umd.edu** in the **pub/amanda** directory.

Acknowledgments

The authors would like to thank Pete Cottrell of the Department of Computer Science, and Steve Miller and Charlie Amos of the Institute for Advanced Computer Studies, for providing the computer facilities on which we tested Amanda, and for putting up with our early, buggy versions. Edward Browdy and Jeff Webber provided early feedback. Harry Mantakos helped find workarounds for some brain dead versions of `ruserok()`, and with other fun porting adventures. Thanks to Staff World for torture testing the software's error handling features. Congratulations to Pete Cottrell for winning our naming contest by coming up with *Amanda*.

Last, but certainly not least, the authors would like to thank Ann Gallagher, Vanessa Chernick, and Kristen Tsapis, respectively, for their moral support and understanding.

References

1. Steve M. Romig, ''Backup at Ohio State, Take 2,'' *Proceedings of the Fourth Large Installation Systems Administration Conference,* pp. 137-141, The Usenix Association, Oct 1990.
2. Rob Kolstad, ''A Next Step in Backup and Restore Technology,'' *Proceedings of the Fifth Large Installation Systems Administration Conference*, pp. 73-79, The Usenix Association, Sep 1991. ,
3. *EXB-8200 8mm Cartridge Tape Subsystem Product Specification,* Exabyte Corporation, Jan 1990.
4. Steve Shumway, ''Issues in On-line Backup,'' *Proceedings of the Fifth Large Installation Systems Administration Conference,* pp. 81-87, The Usenix Association, Sep 1991.
5. Elizabeth D. Zwicky, ''Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not,'' *Proceedings of the Fifth Large Installation Systems Administration Conference,* pp. 181-190, The Usenix Association, Sep 1991.
6. *TLZ04 Cassette Tape Drive,* Digital Equipment Corporation, Apr 1990.
7. Donald K. Friesen and Michael A. Langston, ''Analysis of a Compound Bin Packing Algorithm,'' *Journal of Discrete Mathematics*, vol. 4, no. 1, pp. 61-79, SIAM, February 1991.

Author Information

James da Silva was a high-school hacker in the early '80s. He received a National Merit Scholarship from Georgia Tech in 1983, but soon left school to work in the Real World. He was a Systems Engineer for Electronic Data Systems, writing applications programs first on the PC, then under UNIX. In 1987 he escaped the Real World and headed for the ivory towers of the University of Maryland. He works there as a Research Programmer for the Systems Design and Analysis Group of the CS Department, and is still lingering in search of those last few undergraduate credits. Jaime can be reached at jds@cs.umd.edu .

Ólafur Guðmundsson was born in Reykjavík, Iceland. He graduated from the University of Iceland in 1983 with a B.S. in Computer Science. He worked as a systems programmer on VMS machines at the University of Iceland from 1983 to 1984. In 1984 he joined the graduate program of the Department of Computer Science at the University of Maryland where he had to learn some new operating systems, most of them unpleasant mainframe systems, until discovering UNIX. Ólafur obtained a Masters degree in 1987. Since then he has worked as a Faculty Research Assistant in the Department of Computer Science at University of Maryland. In this position he has been primarily involved in research, development and implementation of a distributed, hard-real-time operating system *Maruti*, but he has also worked on practical problems in computer networks and operating systems. Ólafur can be reached at ogud@cs.umd.edu or ogud@rhi.hi.is.

Daniel Mossé was born in Rio de Janeiro, Brazil, when Brazil won a Soccer World Cup. He did his undergrad in Brasilia, the capital, in Math, and got his MS from the University of Maryland, where he is currently a Ph.D. candidate, soon to graduate. He can be reached through e-mail at mosse@cs.umd.edu. Interests range from anything interesting, to real-time operating systems, resource allocation and scheduling problems, fault tolerance, and databases.

The DIDS (Distributed Intrusion Detection System) Prototype

Steven R. Snapp, Stephen E. Smaha – Haystack Laboratories, Inc.
Daniel M. Teal, Tim Grance – United States Air Force Cryptologic Support Center

ABSTRACT

Intrusion detection is the problem of identifying unauthorized use, misuse, and abuse of computer systems by both system insiders and external penetrators. The growth in numbers and complexity of heterogeneous computer networks provides additional implications for the intrusion detection problem. In particular, the increased connectivity of computer systems gives greater access to outsiders, and makes it easier for intruders to avoid detection. We are designing and implementing a prototype Distributed Intrusion Detection System (DIDS) that combines distributed monitoring and data reduction (through individual Host and LAN Monitors) with centralized data analysis (through the DIDS Director) in order to monitor a heterogeneous network of computers. This approach is unique among current intrusion detection systems. One of the problems considered in this paper is the Network-user Identification (NID) problem, which is concerned with tracking a user moving across the network, possibly with a new user-id on each computer. Initial system prototypes have provided quite favorable results on both the NID problem and the detection of other attacks on a network. This paper provides an overview of the motivation behind DIDS, the system architecture and capabilities, and a discussion about the implementation of the system prototype.

Introduction

Intrusion detection is the problem of identifying individuals who are using a computer system without authorization (i.e., the *external threat*) and those who have legitimate access to the system but are exceeding and/or abusing their privileges (i.e., the *insider threat*). Work is being done in parallel on Intrusion Detection Systems (IDS's) to monitor a single host [9,12,8], several hosts connected by a network [7,6,13], and a broadcast Local Area Network (LAN) [3,4].

The large numbers and complexity of heterogeneous computer networks has serious implications for the intrusion detection problem. Foremost among these implications is the increased opportunity for unauthorized access via the network's connectivity. This problem is intensified when dial-up or internetwork access is allowed, as well as when unmonitored hosts (viz. hosts without audit trails) are present on the network. The use of distributed rather than centralized computing resources also implies reduced control over those resources. Moreover, multiple independent computers generate more audit data than a single computer, and this audit data is dispersed among the various systems. Clearly, not all of the audit data can be forwarded to a single IDS for analysis; some analysis must be accomplished at the local host.

This paper describes a prototype Distributed Intrusion Detection System (DIDS) which generalizes the target environment in order to monitor multiple hosts connected via a network and the network itself. The DIDS components include the DIDS Director, a single Host Monitor per host, and a single LAN Monitor for each LAN segment of the monitored network. Information is gathered and processed locally by each distributed component, with important events and information transported to, and analyzed at, a central location (viz. an Expert System, which is a sub-component of the Director). This architecture provides the capability to aggregate information from numerous different sources. The system is designed to work with any audit trail format as long as certain pieces of critical information are provided by the auditing mechanism.

DIDS is designed to operate in a heterogeneous environment composed of C2 [1] or higher rated computers. The DoD Class C2 (Controlled Access Protection) rating enforces a finely grained discretionary access control that makes users individually accountable for their actions through login procedures, auditing of security-relevant events, and resource isolation. The target environment consists of several hosts connected by a single broadcast LAN segment (presently an Ethernet, see Figure 1). The use of C2-rated systems implies a consistency in the content of the system audit trails. This allows us to develop standard representations into which we can map audit data from UNIX, VMS, or any other system with C2 auditing capabilities. Some abstraction is performed on the raw audit data in order to transform the data into the standard representation. The C2 rating also provides, as part of the Trusted

Computing Base (TCB), the security and integrity of the host's audit records. Although the hosts must comply with the C2 specifications in order to be monitored directly, the network related activity of non-compliant hosts can be monitored via the LAN Monitor. Since all attacks that utilize the network for system access will pass through the monitored segment, the LAN Monitor will be able to analyze all of this traffic.

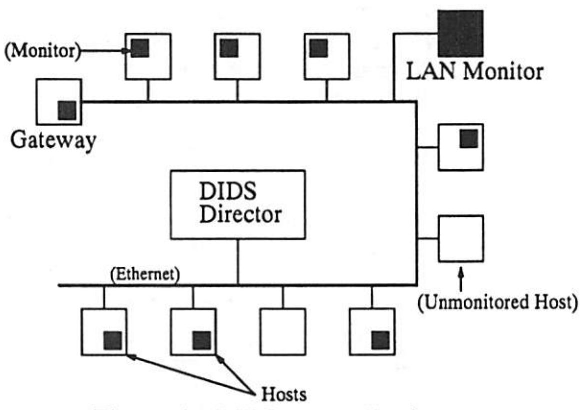


Figure 1: DIDS Target Environment

DIDS Architecture

The DIDS architecture combines distributed monitoring and data reduction with centralized data analysis. This approach is unique among current intrusion detection systems. The major components of DIDS are the *DIDS Director*, a single *Host Monitor* per host, and a single *LAN Monitor* for each broadcast LAN segment in the monitored network (Figure 2).

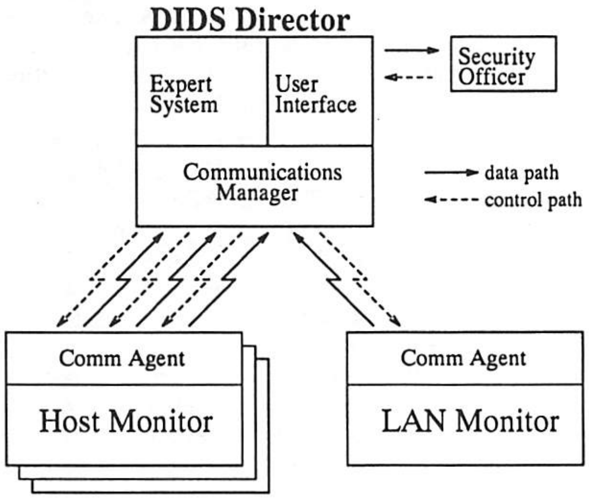


Figure 2: Communications Architecture

DIDS can potentially handle hosts without monitors since the LAN Monitor can report on the network activities of such hosts. The Host and LAN Monitors are primarily responsible for the collection of evidence of unauthorized or suspicious activity, while the DIDS Director is primarily responsible for its aggregation and evaluation.

The Host Monitor collects and analyzes audit records from the host operating system. The audit records are scanned for *notable events*, which are transactions that are of interest independent of any other records. These include, among others, failed events, user authentications, changes to the security state of the system, and any network access such as *rlogin* and *rsh*. These notable events are then sent to the DIDS Director for further analysis. The Host Monitor also tracks user sessions and reports anomalous behavior aggregated over time through user/group profiles. It also searches the event stream for attack signatures, which are sequences of events that are considered to be indicative of attack behavior.

The LAN Monitor's main responsibility is to observe all of the traffic on its segment of the LAN in order to monitor host-to-host connections, services used, and volume of traffic. The LAN Monitor reports on such network activity as *rlogin* and *telnet* connections, the use of security-related services, and changes in network traffic patterns. It is also used to help verify the owners of certain connections between hosts.

Each Host and LAN Monitor has a single *communications agent* that provides an interface between the monitor and the DIDS Director. The agent serves as a buffer between the local monitor and the DIDS Director by handling all communications into and out of the host. This design allows the monitor's analysis components to concentrate on detecting intrusions and not be concerned with communications requirements.

The DIDS Director is divided into three components. The *Communications Manager* is responsible for the transfer of data between the DIDS Director and each Host and LAN Monitor via the local agent. It receives notable event records and system reports from each Host and LAN Monitor, and then sends them to the *Expert System* or *User Interface*. The Expert System is responsible for correlating the information and evaluating the data, and then providing reports on the security state of the monitored system. Based on the reports from the Host and the LAN Monitors, the Expert System makes inferences about the security state of each individual host, and aggregates information to report on the state of the entire system. The DIDS Director's User Interface gives the Computer System Security Officer (CSSO) interactive access to the entire system. The CSSO uses the interface to watch activities on each host, observe network traffic, and request more specific types of information from a monitor.

The Network-user Identification (NID)

One of the most interesting challenges for an intrusion detection system operating in a networked environment is tracking users and objects (e.g., files) as they move across the network. This is required to provide accountability in a networked environment. On single hosts, the user-id/password mechanism provides some degree of user accountability, but this is lost when multiple uncoordinated user-ids may belong to one human user. For example, an intruder may use several different accounts on different machines during the course of an attack. Correlating data from several independent sources, including the network itself, can aid in recognizing this type of behavior and tracking an intruder back to their source. In a networked environment, an intruder often chooses to employ the interconnectivity of the computers to hide his true identity and location. A single intruder may use multiple accounts on different machines to launch an attack, and that behavior can be recognized as suspicious only if one knows that all of the activity emanates from a single source. Detecting this type of behavior requires attributing multiple sessions, perhaps with different account names, to a single source.

This problem is unique to the network environment and has not been dealt with before in the context of user accountability. Our solution to the multiple user identity problem is to create a *Network-user Identification* (NID) the first time a user enters the monitored environment, and then to apply that NID to any further instances of that user. All evidence about the behavior of any instance of the user is then accountable to the single NID. In particular, we must be able to determine if "smith@host1" is the same user as "jones@host2". Since the Network-user Identification problem involves the collection and evaluation of data from both Host and LAN Monitors, examining it is a useful method to understand the operation of DIDS.

The Host Monitor

For the current prototype, the Host Monitor operates on a Sun SPARCstation running SunOS 4.1.1 or later with the Sun Basic Security Module (BSM) package installed. Through the BSM security package, the operating system produces audit records for virtually every raw event on the system. These raw events include file accesses, system calls, process executions, and logins. A VMS version of the Host Monitor is currently under development as well.

The Host Monitor consists of three analysis components that act in parallel (Figure 3). Each analysis component processes the same data (possibly in different formats) in order to make its decisions. One component looks for notable events, another builds and maintains session profiles for users and groups of users, and the third component looks for attack signatures.

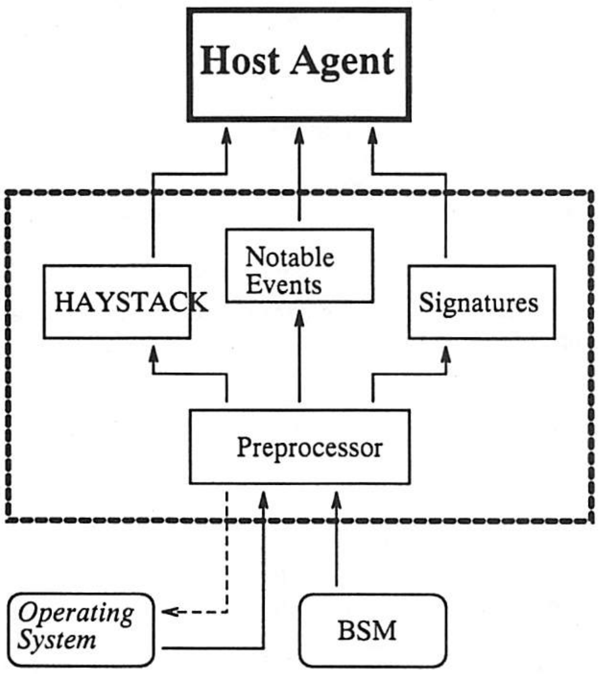


Figure 3: Host Monitor Structure

The Host Monitor *preprocessor* converts the raw audit data generated by the BSM into the abstract events that are processed by each analysis component of the Host Monitor. The preprocessor abstracts from the raw data to remove superfluous information and remove the majority of host specific information. The preprocessor also performs context analysis on each record in order to decide which event type the current record maps into. Since there are more types of raw audit records than abstract events, and the raw audit records appear in different contexts, there is not a simple one-to-one mapping of audit records to abstract events. It is the preprocessor's job to transform each raw audit record into the standard format that the analysis components expect to deal with.

The notable event detector in the Host Monitor examines each event to determine whether or not it should be sent to the Expert System for further evaluation. Certain critical events are always passed directly to the Expert System (i.e., *notable events*); others are processed locally by the Host Monitor in order to generate profiles, and only summary reports are sent to the Expert System. Thus, one of the design objectives is to keep as much of the processing operations at the local host as possible.

Of all possible events generated by the preprocessor, only a subset is sent to the Expert System for further consideration. For the creation and application of the NID, it is those events which relate to the creation or modification of user sessions that are important. Communications events, authentication events, and privilege changes are especially useful for the NID problem. The Host Monitor consults external tables to determine which events should be sent to the Expert System. Because they relate to events rather than to the audit records themselves, the tables and the modules of the Host Monitor which use them are portable across operating systems. The only portion of the Host Monitor which is operating system dependent is the module which creates the abstract events based on audit records.

The HAYSTACK Component

The HAYSTACK component of the Host Monitor reduces the voluminous system event stream to short summaries of user behaviors, anomalous events, and security incidents [9]. In addition to providing this data reduction, HAYSTACK attempts to detect several types of intrusions: attempted break-ins, masquerade attacks, penetration of the security system, leakage of information, denial of service, and malicious use. HAYSTACK's operation is based on behavioral constraints imposed by official security policies and on models of typical behavior for user groups and individuals. HAYSTACK helps to detect intrusions (or misuse) in two different ways.

HAYSTACK may tag particular security subjects and objects as requiring special monitoring. This is analogous to setting an alarm to go off when a particular user-id is active, or when a particular file or program is accessed. This alarm may also increase the amount of reporting of the user's activity.

HAYSTACK performs two different kinds of statistical analysis. The first kind of statistical analysis yields a set of suspicion quotients. These are measures of the degree to which the user's aggregate session behavior resembles one of the target intrusions that HAYSTACK is trying to detect.

About two dozen features (behavioral measures) of the user's session are monitored on the system, including time of work, number of files created, number of pages printed, etc. Given a list of the session features whose values were outside the expected ranges for the user's security group, plus the estimated significance of each feature violation for detecting a target intrusion, HAYSTACK computes a weighted multinomial suspicion quotient that the session resembles a target intrusion for the user's security group. The suspicion quotient is therefore a measure of the anomalousness of the session with respect to a particular weighting of features. HAYSTACK emphasizes that such suspicions are not "smoking guns", but are rather hints or hunches to the security officer or the DIDS Director that may warrant further investigation.

The second kind of statistical analysis detects variation within a user's behavior by looking for significant changes (trends) in recent sessions compared to previous sessions.

The Signature Analysis Component

The attack signature analysis component of the Host Monitor defines attack-type behavior and recognizes sequences of events that closely match predefined signatures [10,11]. Signature analysis searches for known attack sequences, attacks that exploit known flaws or administrative vulnerabilities, and attacks that a masquerader would employ to change the security state of the system, browse through the file system, store information for future use, or attempt to hide his tracks. Signature analysis builds up a context of activity that is based on the users previous events.

Space	Static/ Dynamic	Partitions
Identity	dynamic	super-user normal
Identity	static	super-user normal
Object location	dynamic	read-only system space writable system space owned user space other user space
User location	dynamic	read-only system space writable system space owned user space other user space
Origin	static	physical terminal local host remote host
Event time	dynamic	business hours off hours
System state	dynamic	normal activity suspicious activity under attack
Session security state	dynamic	normal attacker
Security suspicion state	dynamic	normal excessive

Table 1: Signature Analysis Spaces and Partitions

It then looks at (context, event) tuples in an attempt to identify membership in or resemblance to the described set of signatures. It is the context that determines whether or not the current event triggers an instance of a signature; the same event occurring in a different context may not trigger an instance of that signature. Conversely, in many (but not all) cases it is the event that determines the significance of a change in context.

The context consists of a set of spaces (Table 1) that describe various attributes of the user session up to a particular point in time. The information obtained at login and from each abstract audit event serve to build the context. Each space is divided into a set of partitions. After each event has been processed, a user is in at most one partition within each space. A transition between partitions is caused either by an audit event or by external manipulation (e.g., a message from the Expert System in the DIDS Director).

Spaces are either static or dynamic. Static spaces are defined at login; no transitions between partitions occur during a user session. Dynamic spaces are initially defined at login; transitions between partitions occur during a user session. Some dynamic spaces may be "read-only" for the signature mechanism. Those "read only" spaces can only be manipulated through an external mechanism. A transition between partitions of a dynamic space and the association that transition had with the user privilege level is of greatest interest to the signature mechanism.

The LAN Monitor

The LAN Monitor is a subset of UC Davis' Network Security Monitor (NSM) [3,4]. The LAN Monitor builds its own "LAN audit trail". The LAN Monitor observes each and every packet on its segment of the LAN and, from these packets, it is able to construct higher-level objects such as connections (logical circuits), and service requests that use the TCP/IP or UDP/IP protocols. In particular, it audits host-to-host connections, services used, and volume of traffic per connection.

The LAN Monitor uses simple analysis techniques to identify significant events. The events include the use of certain services (e.g., *rlogin* and *telnet*) as well as activity by certain classes of hosts (e.g., a PC without a Host Monitor). The LAN Monitor also uses and maintains profiles of expected network behavior. The profiles consist of expected data paths (e.g., which systems are expected to establish communication paths to which other systems, and by which service) and service profiles (e.g., what a typical *telnet*, *mail*, or *finger* is expected to look like).

The LAN Monitor also uses heuristics in an attempt to identify the likelihood that a particular connection represents intrusive behavior. These heuristics consider the capabilities of each of the network services, the level of authentication required for each of the services, the security level for each machine on the network, and signatures of past attacks. The abnormality of a connection is based on the probability of that particular connection occurring and the behavior of the connection itself. The LAN Monitor is also able to provide a more detailed examination of any connection, including capturing every character crossing the network. This capability can be used to support a directed investigation of a particular subject or object.

The LAN Monitor has several responsibilities with respect to the creation and use of the NID. The LAN Monitor is responsible for detecting any connections related to *rlogin* and *telnet* sessions. Once these connections are detected, the LAN Monitor can be used to verify the owner of a connection. The LAN Monitor can also be used to help track tagged objects moving across the network. The CSSO can also tap into a network connection to closely monitor a suspicious user's behavior.

The Expert System

DIDS utilizes a rule-based (or production) expert system that is written in CLIPS, a C language expert system implementation from NASA [2]. The Expert System uses rules derived from a hierarchical model that describes data abstractions used in inferring an attack on a local area network. That is, it describes the transformation from raw audit data to high level hypotheses about intrusions and about the overall security of the monitored environment. In abstracting and correlating data from the distributed sources, the model builds a virtual machine which consists of all the connected hosts as well as the network itself. This unified view of the distributed system simplifies the recognition of intrusive behavior which spans individual hosts. The model is also applicable to the trivial network of a single computer.

The low level Expert System objects are the abstract event reports provided by the LAN Monitor and the Host Monitor. These reports are both syntactically and semantically independent of the source.

The goal is to introduce a single identification for a user across many hosts on the network. Upper layers of the model treat the network-user as a single entity, essentially ignoring the local identification on each host. Similarly, above this level, the collection of hosts on the LAN are generally treated as a single distributed system with little attention being paid to the individual hosts.

Events are then placed in context. There are two kinds of context: temporal and spatial. As an example of temporal context, behavior which is unremarkable during normal business hours may be highly suspicious during off hours [5]. In addition to the consideration of external temporal context, the Expert System uses time windows to correlate events occurring in temporal proximity. Spatial context implies the relative importance of the source of events. That is, events related to a particular user, or events from a particular host, may be more likely to represent an intrusion than similar events from a different source.

In the context of the Network-user Identification problem we are concerned primarily with the audit data, the event, and the subject. The generation of the first two of these have already been discussed; thus, the creation of the subject is the focus of the following section.

Building the NID

The only legitimate ways to create an instance of a user within UNIX are for the user to login from a terminal, console, or remote source, to change the user-id of an existing instance, or to create additional instances (local or remote) from an existing instance. In each case, there is only one initial login (system wide) from an external device. When this original login is detected, a new unique NID is created. This NID is applied to every subsequent action generated by that user. When a user who already has a NID creates a new login session, that new session is associated with his original NID. Thus the system maintains a single identification for each physical user.

We consider an instance of a user to be the 4-tuple *<session_start, user-id, host-id, timestamp>*. Thus, each login creates a new instance of a user. In associating a NID with an instance of a user, the Expert System first tries to use an existing NID. If no NID can be found which applies to the instance, a new one is created. Trying to find an applicable existing NID consists of several steps. If a user changes identity (e.g., using the UNIX *su* command) on a host, the new instance is assigned the same NID as the previous identity. If a user performs a remote login from one host to another host, the new instance gets the same NID as the source instance. When no applicable NID is found, a new unique NID is created.

There is still some uncertainty involved in attempting to solve the Network-user Identification problem. If a user leaves the monitored domain and then reenters it with a different user-id, the uncertainty is resolved by creating a session "thumbprint". The thumbprinting technique allows us to examine certain characteristics of two different network connections in an attempt to correlate them and determine if two connections belong to the same session. Similarly, if a user passes through an unmonitored host, the need again arises to use the thumbprinting technique in an attempt to match a connection entering the host with a connection leaving the host. Multiple connections originating from the same host at approximately the same time also creates uncertainty if the user names on the target hosts do not provide any helpful information. The Expert System can make a final decision with additional information from the Host and LAN Monitors that can (with high probability) disambiguate the connections.

Conclusion

Our Distributed Intrusion Detection System (DIDS) addresses the shortcomings of current single host IDS's by generalizing the target environment to incorporate multiple hosts connected via a local area network (LAN). Most current IDS's do not consider the impact of the LAN structure when attempting to monitor user behavior for attacks against the system. Intrusion detection systems designed for a network environment will become increasingly important as the number, size, and complexity of LAN's continue to increase. Our prototype has demonstrated the viability of our distributed architecture in solving the Network-user Identification problem. We have tested the system on a network of Sun SPARCstations and it has correctly tracked network users in a variety of scenarios. Work continues on the design, development, and refinement of rules, particularly those which can take advantage of knowledge about specific kinds of attacks. In addition to the current Host Monitor, which is designed to detect attacks on general purpose multi-user computers, we intend to develop monitors for application specific hosts such as file servers and gateways. In support of the ongoing development of DIDS we are planning to extend our model to a hierarchical Wide Area Network environment.

Acknowledgments

The DIDS project is a joint effort between the United States Air Force Cryptologic Support Center, Haystack Labs, the University of California at Davis, and Lawrence Livermore National Labs.

References

1. Department of Defense, *Trusted Computer System Evaluation Criteria,* National Computer Security Center, DOD 5200.28-STD, Dec. 1985.
2. J. C. Giarratano, *CLIPS User's Guide Volume 1 - Rules,* NASA Lyndon B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, Houston, TX, January 1991.
3. L. T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A Network Security Monitor," *Proc. 1990 Symposium on Research in Security and Privacy*, pp. 296-304, Oakland, CA, May 1990.
4. L. T. Heberlein, K. N. Levitt, and B. Mukherjee, "A Method to Detect Intrusive Activity in a Networked Environment," *Proc. 14th National Computer Security Conference*, pp. 362-371, Washington, D.C., Oct. 1991.
5. T. Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey," *Proc. 11th National Computer Security Conference*, pp. 65-73, Baltimore, MD, Oct. 1988.

6. T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H. S. Javitz, A. Valdes, and P. G. Neumann, "A Real-Time Intrusion-Detection Expert System (IDES)," Interim Progress Report, Project 6784, SRI International, May 1990.
7. T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, and C. Jalali, "IDES: A Progress Report," *Proc. Sixth Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 1990.
8. M. M. Sebring, E. Shellhouse, M. E. Hanna, and R. A. Whitehurst, "Expert Systems in Intrusion Detection: A Case Study," *Proc. 11th National Computer Security Conference*, pp. 74-81, Oct. 1988.
9. S. E. Smaha, "Haystack: An Intrusion Detection System," *Proc. IEEE Fourth Aerospace Computer Security Applications Conference*, Orlando, FL, Dec. 1988.
10. S. R. Snapp, "Signature Analysis and Communication Issues in a Distributed Intrusion Detection System," MS Thesis, Division of Computer Science, University of California, Davis, Aug. 1991.
11. S. R. Snapp, B. Mukherjee, and K. N. Levitt, "Detecting Intrusions Through Attack Signature Analysis," *Proc. Third Workshop on Computer Security Incident Handling*, Herndon, VA, Aug. 1991.
12. H. S. Vaccaro and G. E. Liepins, "Detection of Anomalous Computer Session Activity," *Proc. 1989 Symposium on Research in Security and Privacy*, pp. 280-289, Oakland, CA, May 1989.
13. J. R. Winkler, "A UNIX Prototype for Intrusion and Anomaly Detection in Secure Networks," *Proc. 13th National Computer Security Conference*, pp. 115-124, Washington, D.C., Oct. 1990.

Author Information

Steven R. Snapp received a Bachelor's Degree in Computer Science from Colorado State University in 1989, and a Master's Degree in Computer Science from the University of California at Davis in 1991. He has been working on the DIDS project since its inception in 1990, and is currently a software engineer at Haystack Labs in Austin, Texas. Steve was the lead author of an earlier paper on DIDS that won an Outstanding Paper Award from the 14th National Computer Security Conference held in Washington, D.C. in October 1991. Steve can be reached via electronic mail at Snapp@Dockmaster.NCSC.mil.

Stephen E. Smaha is the President of Haystack Labs, Inc. He is the designer and implementer of the HAYSTACK Intrusion Detection System, which is in use at Air Force bases and other U.S. Government sites. Steve is the architect of the DIDS system, and has lectured widely on computer security and intrusion detection. He has graduate degrees in Computer Science and Philosophy from Rutgers University and the University of Pittsburgh, and a BA from Princeton. Steve can be reached via electronic mail at Smaha@Dockmaster.NCSC.mil.

Lt. Daniel M. Teal received a Bachelor's Degree in Electrical Engineering from the Massachusetts Institute of Technology in 1989. He was commissioned by the Air Force and assigned to the AFCSC as a Communications and Computer Security Engineer. He is the lead technical program manager for the DIDS project, and serves as a technical consultant on UNIX security for his directorate. Dan can be reached via electronic mail at Teal@Dockmaster.NCSC.mil.

Captain Tim Grance is a computer scientist for the Air Force Office of Communications-Computer Systems Security located at the Air Force Cryptologic Support Center, Kelly Air Force Base, Texas. He works in the Engineering Division Counter Measures Development Branch as the DIDS project manager. His primary investigative areas are intrusion detection, network security, and UNIX security. He has over 11 years experience in computers and security, and holds a Bachelor of Science degree in Computer Science from the Georgia Institute of Technology, Atlanta, Georgia. Tim can be reached electronically at Grance@Dockmaster.NCSC.mil.

A Privilege Mechanism for UNIX System V Release 4 Operating Systems

Charles Salemi, Suryakanta Shah, Eric Lund – UNIX System Laboratories, Inc.

ABSTRACT

Any multi–user, multi–tasking operating system, such as the UNIX SVR4 Operating System, must provide protection mechanisms that prohibit one user from interfering with another user, or limit the execution of certain system operations that affect critical system resources. These protection mechanisms must also provide the ability to override these restrictions, commonly referred to as *privilege*. For over twenty years, UNIX–based operating systems have had one such privilege, called ''root'' or ''super–user'' which is signified by a process whose effective user ID is 0. The ''super–user'' has the ability to override the restrictions imposed by these protection mechanisms. In the UNIX System V Release 4 Enhanced Security product this single, omnipotent, privilege is divided into a set of discrete privileges designed to assure that sensitive system services execute with the minimum amount of privilege required to perform the desired task.

This paper describes the *privilege control mechanism* implemented as part of the UNIX System V Release 4.1 Enhanced Security (SVR4.1ES) product. The SVR4.1ES privilege control mechanism separates the privilege mechanism from the access control mechanism, it provides for fine grained control over sensitive operation access by users, and it controls the propagation of privilege from one process to another. Our goals also include accommodating multiple privilege control mechanisms within the UNIX System V kernel. These privilege mechanisms can be ''plugged'' into the kernel through well defined interfaces, much the same way as UNIX file systems are currently added to the kernel.

Introduction

The granularity of the ''root'' privilege is too coarse for a system that dictates fine granularity in the assignment of privilege and the ability to control the assertion of privilege throughout the execution of a process. The SVR4.1ES privilege control mechanism is designed to meet the the following goals:

- to make the privilege control mechanism a separate, loadable module,
- to make minimal changes to existing kernel code,
- to separate the privilege control mechanism and access mechanism,
- to make the file–based privileges file system independent,
- to preserve UNIX System V compatibility.

Our approach was to use the concept of privilege sets that are assigned to both processes and executable files. The concept of privilege sets is not new. We are aware of two papers discussing the implementation of privilege control mechanisms that also made use of privilege sets [1][2].

The feature that makes the implementation of our privilege control mechanism unique is our fourth goal: file system independence. This feature makes it easy for file system developers to make use of our privilege mechanism. In addition, existing file system types function properly without any modifications. This paper describes the kernel interface routines we defined that are required to support the concept of separate privilege modules. It also describes the major differences between the two privilege modules supported under SVR4.1ES.

Problem Definition

Before we begin our discussion, it is necessary to provide a definition for the term privilege. Simply stated, ''privilege'' is the ability to override restrictions imposed by protection mechanisms. For example, a process[1] requires privilege to change the system date, mount or unmount a file system, or modify file attributes (if not the owner of the file) because these operations are restricted by the protection mechanisms.

Our mission from the start was explicit: add the capability of supporting discrete privileges assigned to each sensitive system operation in the kernel. This was necessary to provide a privilege policy that based propagation of privilege on attributes other than the effective user ID. However, we also had the requirement to remain compatible with previous versions of the UNIX Operating System by maintaining the concept of a ''super–user.''

[1]The term *process* is defined in [3] as an instance of an *executable file* in execution. An executable file is defined to be a *program*.

The privilege control mechanism we chose for UNIX SVR4.1ES removes the omnipotence of effective user ID 0 by defining a set of discrete privileges, each one used to override individual restrictions imposed by *sensitive* system operations. It also defines an inheritance policy, by associating privileges with executable files, to control propagation of privileges. In addition, it gives flexibility to designers of privilege modules by providing well–defined interface routines that are general enough to support a privilege policy based on any process attribute.

Defining the Kernel Privilege Interface

Why a Loadable Privilege Control Mechanism?

Our first goal was to make the privilege control mechanism "plug compatible" so administrators could choose the privilege policy they preferred at system configuration time.

There were quite a few "religious" debates surrounding this particular goal. One school of thought argued very strongly that the operating system should be configured with a privilege policy that was in effect for the entire life of the system. It was felt that any deviation from the privilege policy only meant that the potential existed for more things to go wrong.[2] Another school of thought disagreed contending that the system went through one or more phases before the entire security policy was in place. They felt that while the system was transitioning to the final phase, the privilege policy should be based on the effective user ID and once the transition to a secure system was complete, change the privilege policy to one that was file–based.

We decided to go with the first interpretation for several reasons:

- The privilege mechanism has always been a "point of attack" with respect to security break–ins. Adding complexity for determining which privilege policy was in effect appeared to weaken the privilege mechanism.
- Using the modular approach and defining separate privilege policies does not preclude the use of a privilege control module that behaves in the manner described by the second interpretation.
- It is flexible enough to allow a designer of a privilege module the option of basing the privilege policy on other attributes of the process such as the effective or real group ID, the sensitivity label of the process, etc. with minor modification to existing kernel source code. In this case, only the privilege module source code would require modification. The *privilege request* checks in the kernel would remain undisturbed.

[2]We are all acutely aware of *Murphy's Law.*

Therefore, to accomplish this goal, a general privilege interface was required that would allow for the flexibility of specifying the privilege policy desired for a particular privilege module.

General Privilege Interface

Each concept below has a corresponding routine in a specific privilege control mechanism. The behavior for each routine is relative to the privilege policy enforced by that privilege mechanism.

Initialization

To begin with, there must be a way to initialize the system privilege mechanism at system initialization time. Functions that might be performed by this procedure include: allocation of data structures, installation of "bootstrap" data, or initialization of local static variables.

Privilege Requests

A procedure is needed to determine if a privilege requested by a process is contained in the set of privileges currently in effect for that process. This procedure is the *policy maker* of any privilege module because it makes the decision to grant or deny privilege when a request is made.

Propagation

Another procedure is required to calculate the privileges for new processes. This procedure provides the privilege propagation model for the module.

Process Privilege Manipulation

A procedure is required to allow a process to count, set, clear, and retrieve its privilege sets.

Process Privilege Recalculation

A procedure is required that will recalculate process privileges whenever the effective user ID is modified.[3]

File Privilege Manipulation

Another procedure is required to assign and retrieve privilege associated with a file. This allows software external to the policy module to identify files that are included in the system. The system is made up of files that have been analyzed and determined to securely use certain privileges.

When retrieving privileges this procedure must retrieve them from the same data structure where they are stored for use by the propagation procedure.

File Privilege Removal

A routine is required to remove the privilege information associated with system files whenever media containing "trusted" files is removed from the system. This prevents the introduction of "Trojan Horses" when a new file system is mounted on the same mount point.

[3]This procedure is required to maintain compatibility with ID–based privilege modules.

Privilege Mechanism Information

A procedure is required to allow a process the ability to retrieve specific information regarding the privilege module. This information might be in the form of how many privilege sets are supported by the privilege mechanism, what the names of those sets are, the number of privileges contained in each set, which privilege mechanism is in effect, etc. Currently, two privilege modules exist that make use of the privilege interface routines. They are:

SUM module provides a privilege policy that supports a set of discrete privileges and the "super-user" concept. This module is considered to be ID-based since the propagation of privilege is based on the effective user ID of the process.

LPM module provides a privilege policy that supports a set of discrete privileges and an inheritance policy.[4] This module is considered to be *file-based* since the propagation of privilege is based on the inheritance policy that uses the maximum privilege set of the current process and the set (or sets) of discrete privileges assigned to the executable program file being `exec`'ed.

Definitions

The following is a list of the privilege sets required and their definitions. Also indicated is which privilege sets are used by the two privilege modules supported in SVR4.1ES.

Process Privilege Sets

A process can have several privilege sets.[5] These are used to control the assignment of privilege throughout the life of the process. Currently, both privilege modules support the following process privilege sets:

- **Maximum privileges**: the set of privileges held in trust for a process. This is the set of privileges required by a process to complete all of the program tasks.
- **Working privileges**: the set of privileges necessary to complete a particular program task in a process.

The following rules apply to the maximum and working process privilege sets:

- The working set is always a subset of the maximum set.
- The working set may be altered at any time. The new working set must be a subset of the maximum set.
- The maximum set may be altered at any time. The new maximum set is a subset of the previous maximum set.

  When the maximum set is altered, privileges in the working set that are not in the new maximum set are removed from the working set.
- Process privilege sets are unchanged during a `fork`(2) operation. The privilege sets of the child process are identical to those of the parent.

File Privilege Sets:

A file can have several privilege sets.[6] File privilege sets are used to establish the privileges of the *new*[7] process when the file is executed. The following file privilege set is currently supported for both privilege modules:

- **Fixed privileges**: a set of privileges always given to the new process independent of the process privileges of the invoking process.

The following file privilege set is supported only in the *file-based* privilege module:

- **Inheritable privileges**: a set of privileges given to the new process only if the privilege was in the maximum set of the invoking process.

The following rules apply to file privilege sets:

- File privilege sets can only be assigned to ordinary, executable file types.
- All privileges associated with a file are removed when the file is modified.

Isolation of the Privilege Mechanism Code in the Kernel

The second and third goals involve the isolation of the privilege kernel mechanism in the kernel code. The privilege mechanism was *tightly-coupled* with the access mechanism because a process with effective user ID 0 had unlimited access. This association had to be severed. When we did this, however, we wanted to keep the modification of existing kernel source code to a minimum.

Minimizing Kernel Changes

To achieve the second goal, minimizing kernel changes, the kernel source code was analyzed. Two mechanisms for determining privilege in the kernel were identified:

1. calling the internal kernel routine `suser`() to check if the effective user ID was 0, and
2. explicitly checking whether or not the effective user ID was 0.

Existing routines in the kernel that either called the `suser`() routine or checked the effective user ID

[4]Defined in the *privilege propagation* interface routine.

[5]The current implementation limit is 255.

[6]File privilege sets are also limited to 255.

[7]The term *new* is used here rather than child because a process may be *exec*'ed directly over the calling process without invoking the `fork`(2) system call. The privilege control mechanism will work properly because the privilege setting for the new process is done before the new process is executed.

had to be modified to call the *privilege request* routine defined in the privilege module.[8]

Separation of Privilege and Access Mechanisms

Our third goal, separation of the privilege mechanism in the kernel, does not affect the current *setuid* and *setgid* mechanism. A process using the *setuid/setgid* mechanism works as it currently does by only allowing a user access to files that they might not normally have based on the file access bits. Removing the ''omnipotence'' of user ID 0 in the kernel means that the file access permission works exactly the same for user ID 0 as any other user ID.

Associating Privilege with a File

Privilege needs to be associated with files that are part of the system. One method of associating privilege with a file is to store the privilege in the *inode* of the file. This method provides a protection from users because of the well–defined system interface for accessing the *inode*. However, this implementation is limited because file system types already in use can not take advantage of a modular privilege implementation.

Because of this limitation another method was needed to achieve the fourth goal of file system independence. This method must provide similar protection from users that the *inode* scheme provides, i.e., it should be accessed only through a well–defined interface. Also, to achieve the first goal, it has to fit well into the modular privilege interface.

To meet these goals a memory–based kernel table is used to define the privileged commands in the operating system. Using a memory based table provides the following benefits:

- File system independence. The privilege control mechanism can work across file system types.
- The number of privileged commands can be reduced, or expanded, independent of the privilege policy in use.
- Privilege can never be *imported* from other media since the privilege information is not part of the file attributes.

ID–Based Mechanism Feature

Using the *setuid–on–exec* mechanism[9] and setting the owner of an executable file to *root* is still supported in the ID–based privilege mechanism because that privilege policy supports the concept of ''super–user.'' However, this mechanism also supports *fixed* privileges on executable files. Therefore, it is possible to assign to an executable file only those privileges *required* to complete all its tasks instead of giving it *all* privileges via the *setuid–on–exec* mechanism.

Compatibility

Our fifth and final goal was to preserve compatibility with previous releases of the UNIX Operating System. As mentioned previously, there are two privilege modules supported in SVR4.1ES.

Most of the routines defined for each privilege module are the same because of the similarities between the two privilege policies. The major differences occur in the *propagation* routine and the *process privilege recalculation* routine.

Privilege Propagation

The *propagation* routine determines how privilege is propagated for the particular privilege policy in use.

File–Based Propagation Model

The following model is in effect for the *file-based* privilege module:

> A privilege can be acquired only if the privilege exists in the *maximum set* of the *calling* process or is in the *fixed set* of a file. A process with an empty *maximum set* can **never** pass a privilege to another process. The computation of the new *maximum* and *working* sets is done in the `exec()` kernel code:

Step [A] The *maximum* set of the calling process is **intersected** with the *inheritable* set associated with the program file being executed.

Step [B] The result of Step [A] is **union**ed with the *fixed* set associated with the program file being executed to form the new *maximum* and *working* sets.

> There is an important principle implemented by this feature: A process gains only those privileges that were in either its fixed or inheritable privileges. A starting process cannot **force** a privilege onto or **through** a new process. If a process mistakenly executes the wrong new process (i.e., a ''Trojan Horse''), it cannot pass any privileges that the new process was not designed to enforce.

Figure 1 illustrates the file–based propagation model.

ID–Based Propagation Model

The following model is in effect for the ID–based privilege module:

> A privilege can be acquired only if the privilege exists in the *maximum set* of the *calling* process or is in the *fixed set* of a file. The computation of the *maximum* and *working* sets for the resulting process is done in the *exec()* kernel code:

[8]The check for privileges in device drivers had already been addressed in SVR4 by providing the DDI/DKI interface routine, `drv_priv` [4].

[9]U. S. Patent # 4,135,240.

First, the maximum privilege set of the calling process is stored in a temporary privilege set when the *propagation* routine is entered. Then, the following conditions are evaluated to determine the maximum and working privilege sets for the resulting process:

Step [A] Does the executable file being **exec**'ed have the *setuid–on–exec* bit asserted? If yes, go to Step [B]. If no, set the temporary privilege set to 0 and go to Step [D].

Step [B] Is the owner of the executable file being **exec**'ed "root"? If yes, turn on all privileges in the temporary privilege set and go to Step [D]. Otherwise, go to Step [C].

Step [C] Is the effective user ID of the calling process "root"? If no, turn off all privileges in the temporary privilege set and go to Step [D]. Otherwise, set an indicator for use later and go to Step [D].

Step [D] Does the executable file being **exec**'ed have any *fixed* privileges? If so, add these privileges to the temporary privilege set.

If the privilege sets for the calling process differ from the temporary privilege set generated above, do the following:

a. set the maximum privilege set for the resulting process to the temporary privilege set.
b. If the indicator was set in Step [C], set the working privilege set for the resulting process to the fixed set of privileges found on the executable file.[10] Otherwise,

[10]This was done to maintain compatibility with older versions of the UNIX operating system.

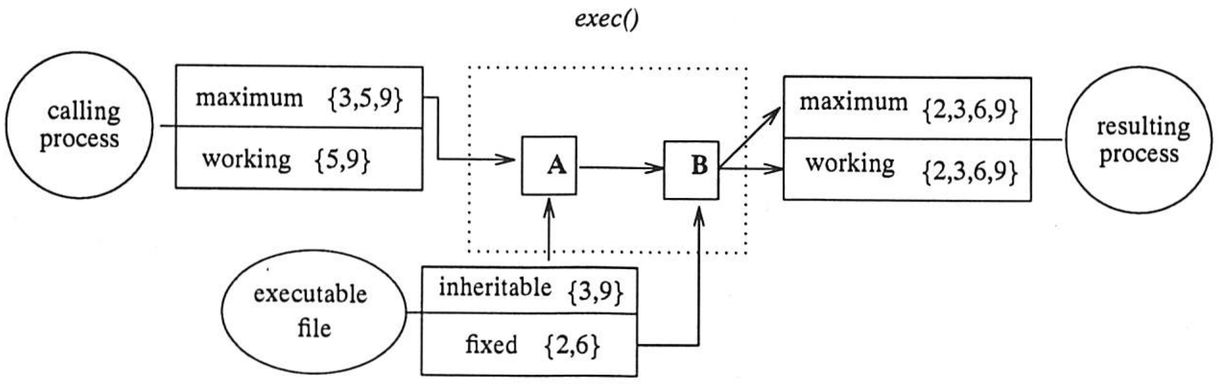


Figure 1: The File–Based Propagation Model

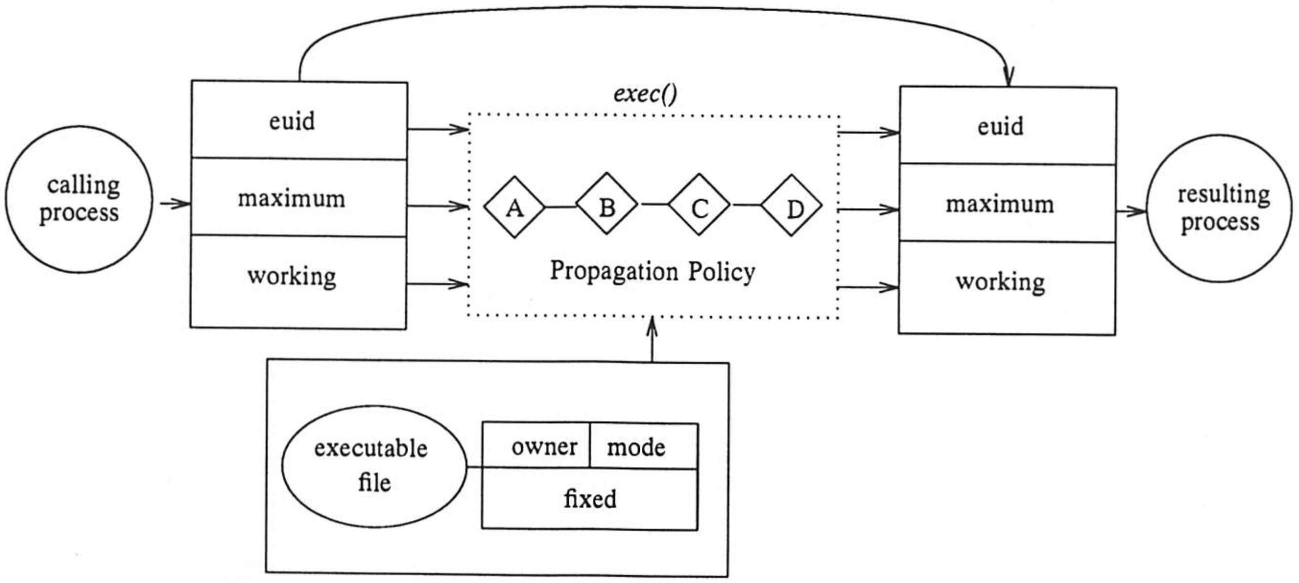


Figure 2: The ID–Based Propagation Model

set the working privilege set for the resulting process to 0.

Figure 2 illustrates the ID–based propagation model.

Privilege Recalculation

The *privilege recalculation* routine adjusts the maximum and working privilege sets of a process according to the privilege module in use.

File–Based Recalculation Model

This routine has a null effect in the file–based privilege module. This is because we intentionally separated the access mechanism from the privilege mechanism (see Section 3.2).

ID–Based Recalculation Model

This routine has extreme significance in the ID–based privilege module because of the tight-coupling between the access and privilege mechanism. This routine is called by the `access()`, `setuid()`, and `seteuid()` system calls.

The following model is in effect for the ID–based privilege mechanism:

The maximum and working privilege sets for the current process are adjusted whenever the effective user ID is modified based on the following conditions:

Condition [A] Clear the maximum and working privilege sets for the current process if none of the process UIDs (effective UID, saved UID, or real UID) equal the *privileged*[11] ID.

Condition [B] Otherwise, set the working privilege set to the maximum privilege set if the effective UID is equal to the *privileged* ID.

Otherwise, only clear the working privilege set if neither Condition [A] nor Condition [B] are true.

Figure i illustrates the ID–based recalculation model.

Conclusion

Moving the privilege mechanism from the kernel proper to a separate, loadable module was extremely simple. We were fortunate that the checks for privilege in the kernel were somewhat well–defined. We were also fortunate that our system was based on UNIX System V Release 4 since a lot of the ground work to make the separation easier was done in that release.

In addition, we maintained compatibility with SVR4 despite extensive modifications required in the kernel. This means that any operating system configured with the SUM module behaves in the exact same manner as an SVR4 Operating System with hard–coded privilege checks for effective UID 0.

[11]The *privileged* ID has traditionally been user ID 0. However, this is now a tunable variable allowing for any user ID to be considered ''all–powerful'' in an ID–based privilege mechanism. User ID 0 is the default value for this variable.

References

[1] Knowles, F. and Bunch, S., *A Least Privilege Mechanism for UNIX*. Proceedings of the 10th National Computer Security Conference. (Sep 1987) Baltimore, MD.

[2] Hecht, M. S., et al. *UNIX Without the Superuser*. Summer USENIX Technical Conferences and Exhibition. (Jun 1987) Phoenix, AZ.

[3] Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice-Hall, Inc., 1986

[4] UNIX Press Title, *UNIX System V Release 4 Device Driver Interface/Driver–Kernel Interface Reference Manual,*

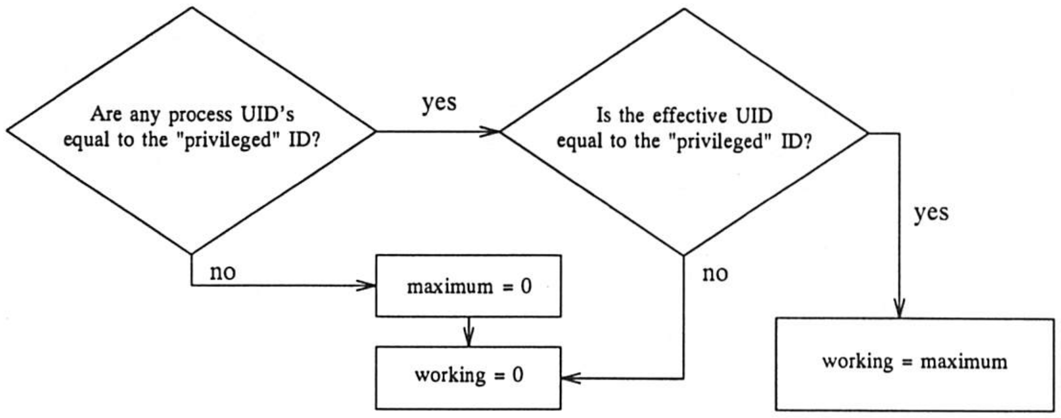

Are any process UID's equal to the "privileged" ID?
yes
Is the effective UID equal to the "privileged" ID?
yes
no
maximum = 0
no
working = 0
working = maximum

Figure 3: The ID–Based Recalculation Model

Author Information

Charles Salemi is currently a Member of Staff at UNIX System Laboratories, Inc. He has been involved with the UNIX Operating System since he received his degree in Computer Science from the City University of New York in 1977. He has been a member of the Security Development Team since 1987. Within this project his principal interests have been in the area of the privilege mechanism and the Identification and Authentication facility. Along with his other accomplishments, he was one of several people responsible for the development of the Source Code Control System (SCCS).

Suryakanta Shah is a Senior Member of Staff at UNIX System Laboratories. She received a MS in Computer Science from Queens College in 1983. During the past 10 years at UNIX System Laboratories, she has worked on numerous features of System V including B2 security, process management, virtual memory, and the RFS distributed file system. Currently, Kanta is adding multiprocessing capabilities to the B2 security features.

Eric Lund received a BA in English from Tufts University in 1984. Since Spring of 1991 he has been a Senior Programmer / Analyst developing on Multi-Level Security features for Cray Research Inc. in Eagan, Minnesota. Prior to his work at Cray, Eric worked at USL where he helped develop the Privilege, Trusted Path, and Audit mechanisms, developed the Trusted Facility Management mechanism, and performed Covert Channel Analysis on System V Release 4.1ES. Eric was also one of the principal developers of the System V Verification Suite. In his spare time, Eric enjoys camping, rock climbing, juggling, woodworking, and beer brewing/tasting.

TCP/IP and OSI Interoperability with the X Window System

Nancy Crowther, Joyce Graham – IBM Cambridge Scientific Center

ABSTRACT

Network users are faced with the problem of making the transition from TCP/IP applications to the emerging Open Systems Interconnection (OSI) protocols. To accomplish this goal, these users must rewrite their code to use OSI, or switch to new applications, or use a gateway between TCP/IP and OSI based applications. This paper details how this problem was solved in work at IBM's Cambridge Scientific Center for one distributed application, the X Window System, and how the same methods could be used for other applications. The draft ANSI standard mapping X to OSI is explained. The changes that were made to the X Window System to support OSI and an X TCP-OSI Gateway are described. The best method for migrating applications was found to be extensions to the socket to support OSI at multiple layers.

Introduction

Interoperability on a network connecting different types of multi-vendor systems is a feature increasingly demanded by users. The X Window System, Transmission Control Protocol/Internet Protocol (TCP/IP), and Open Systems Interconnection (OSI) protocols all promote this interoperability. X allows the workstation user to display results from applications running on multiple heterogeneous systems. Both TCP/IP and OSI protocols are non-proprietary ways to connect these multi-vendor systems, but users who want to change from TCP/IP to OSI in order to use new OSI applications face the problem of converting their current TCP/IP applications to run on the OSI network.

This paper examines one TCP/IP-based application, the X Window System, and explains how we moved it into an OSI environment in experimental prototypes at IBM's Cambridge Scientific Center. X over OSI is a natural pairing to address requirements for enhanced interoperability, but until recently has not been attempted. We explain the mapping of X to OSI as defined in the draft ANSI standard [ANSI91], and describe how we implemented it in two different ways on a UNIX based operating system, IBM's AIX 3.1. Our experience indicates that there are many advantages to modifying the OSI socket programming interface found in 4.3 Reno BSD [BSD43] to support OSI at the Application Layer. We also describe an X TCP-OSI Gateway, for use in networks containing some TCP/IP-based systems and some OSI-based systems.

X Window System

The X Window System is a portable network-transparent window system, allowing multiple applications, called X clients, to run on varied heterogeneous systems and architectures throughout a network and display on any workstation running an X server application. The X server controls the workstation's bitmap display, keyboard and mouse. IBM ships X on most of its platforms, either client application libraries, server, or both, depending on the system.

Clients and server communicate by means of the X protocol, which consists of requests from the client for various functions such as drawing, replies from the server to these requests, events which notify the client of mouse or keyboard input, and error notifications. The X protocol in turn rides on top of a reliable byte stream between client and server. If client and server are on the same system, this reliable byte stream is simply some local interprocess communication mechanism. When client and server are on different systems connected by a network, the reliable byte stream is provided by some communications protocol, typically TCP/IP. All of IBM's current X products use TCP/IP as the underlying network communication protocol.

Figure 1 shows the various parts of an X Window System.

X client application programs use various "toolkits", or application programming interfaces (API's) which implement graphical functions. The toolkits in turn call routines in the various X libraries supplied by the X Window System. The lowest level X library, referenced eventually by all higher layers, is the Xlib, or X library. Buried in this library are the routines which perform the network communication. These communication routines, as distributed in the X source code, currently support TCP/IP and DECnet only. The window manager is also an X client, and it uses the Xlib to perform network communication to the X server. A client in frequent use is the *xterm* program, a terminal emulator. This program displays in a window as if it were the system console. All other programs which write to the console and read from the keyboard can be run "in" this window, sending their output to *xterm*, which in turn communicates through Xlib to the X server.

The X clients and the X server may be executing on the same workstation, or on completely different systems from multiple manufacturers. Since the X protocol is system independent, clients may be written with assurance that they can display on any workstation implementing a standard X server. Similarly, a standard X server can expect to be able to be used by any standard X client from any software vendor. This of course contributes greatly to the goal of interoperability. Applications written for a proprietary windowing system or other user interface are very limited in their use.

Although the X protocol is designed to be able to be implemented from its documentation, in practice vendors use the sample implementation source code, which is available for free from the Massachusetts Institute of Technology X Consortium, and modify it for their own systems. The X source code (in the C language) is written in such a way that it can be compiled for many different operating systems by selecting certain compilation options. The X code is truly portable. Although it was implemented first on 4.2 BSD it runs on operating systems as diverse as VM, MVS, and OS/2, as well as UNIX based operating systems such as AIX. In the current Release 5 from the X Consortium, code which specifically supports AIX 3.1 and one of the graphics adapters for the RISC System/6000 is included. (This code was written by IBM and donated to the X Consortium.) This support is thus freely available to RISC System/6000 users even though it is not yet available as an IBM product.

OSI

While TCP/IP protocols, whose development began in the mid-1970s funded by the Defense Advanced Research Projects Agency (DARPA) for its collection of networks, are the current de facto standard, the long-term replacement for these protocols is the OSI protocols. This suite consists of seven layers of international standard protocols being

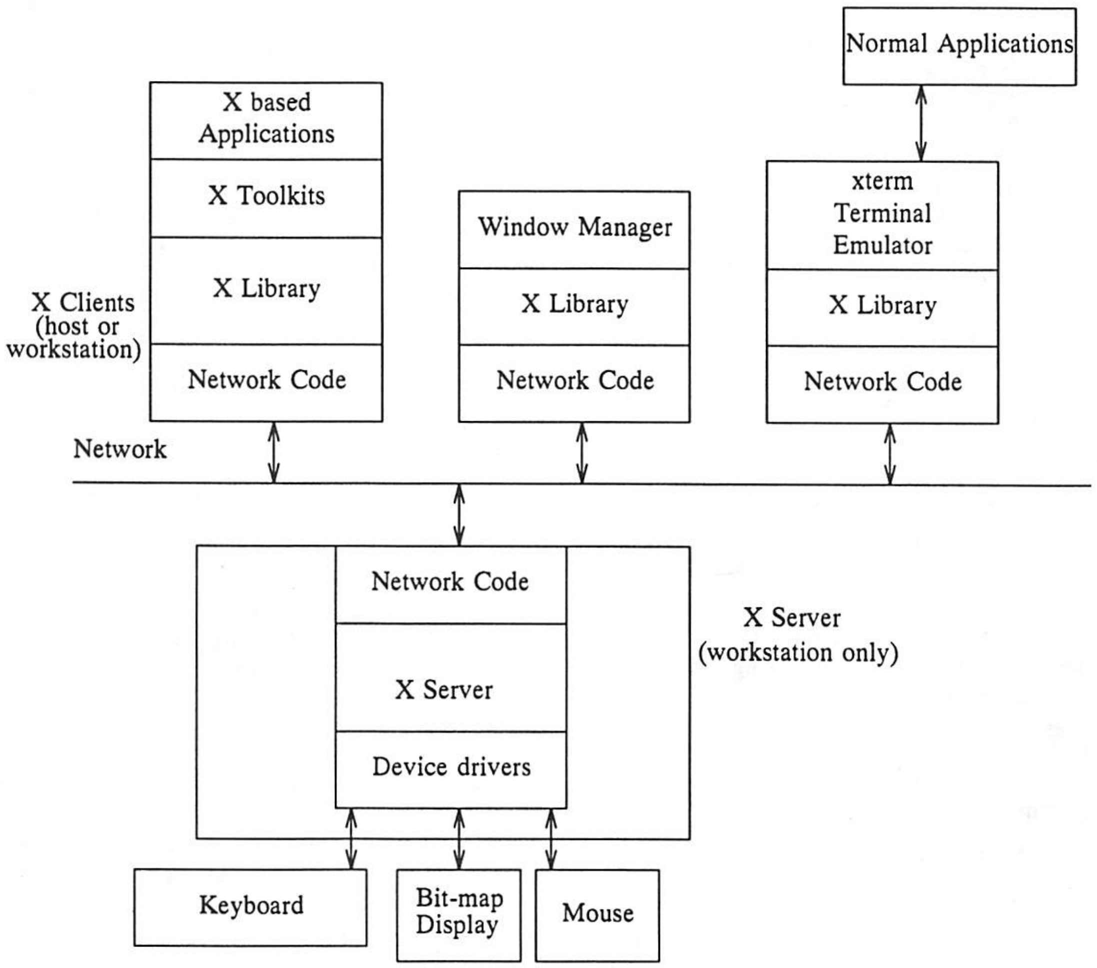


Figure 1: X Window System Parts

developed by the international community to provide both the political and technical solution to worldwide networking. Since August 15, 1990, the US Government has required that new network procurements and major upgrades to existing networks support the Government OSI Profile (GOSIP). GOSIP is a selected subset of the OSI protocols [GOSIP88].

OSI is a complete family of protocols, separating the functions of communication between applications into well-defined layers. Figure 2 shows the OSI reference model [IS7498]. There are two end systems communicating with each other, connected by any number of intermediate systems.

At the highest layer, the Application Layer, applications such as file transfer, mail, and library information retrieval exchange information organized into previously agreed upon data structures. These applications use common application services. One of these is the Association Control Service Element (ACSE). The ACSE manages the connection (called an "association") between the communicating systems. The two sides agree on which application services will be used in the association by exchanging the "application context." The two applications agree on the semantics of the data which is being transferred, but the representation of these data structures on the respective systems may be very different. As simple examples of such differences, one may use ASCII to represent characters and the other may use EBCDIC. One may use the "big endian" method of representing integers (most significant byte first), and the other may use "little endian." The applications are not concerned with these differences; they call on the Presentation Layer to take care of exchanging the data in such a way that the two systems can understand each other. The data structures are specified in a system independent language capable of representing the abstract syntax which defines the data structures exchanged by the communicating applications. The most commonly used abstract syntax language is called Abstract Syntax Notation One (ASN.1), but this is not the only possibility.

The Presentation Layer transforms the local data structures into a system independent syntax (the transfer syntax) which determines the format and ordering of the bits which are actually sent over the network to the other side. The receiving Presentation Layer decodes from the transfer syntax into its own local forms of data representation. The Presentation Layer on each side is able to do this because it has knowledge about which transfer syntax was used to encode the data, and which abstract syntax was used by the Application Layer. The pair of

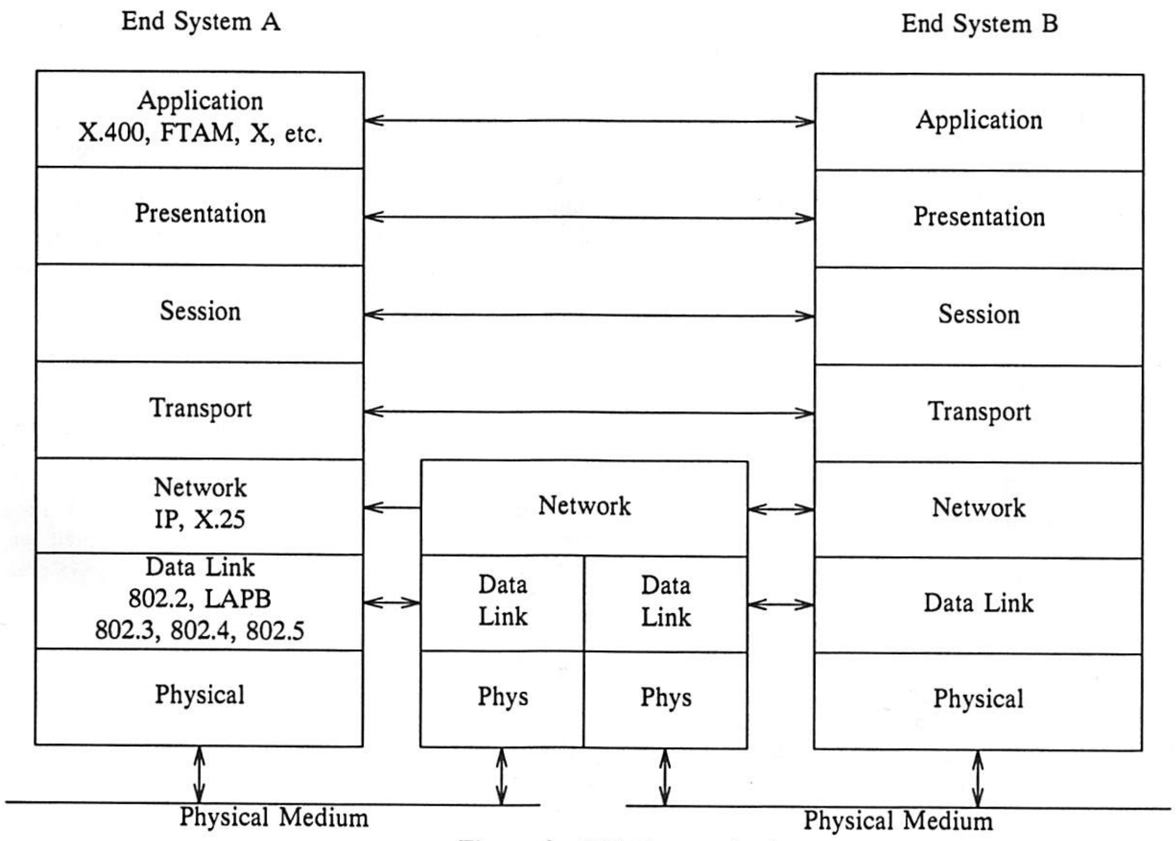


Figure 2: OSI Communication

syntaxes to be used in the communication, called the "presentation context", is negotiated and agreed upon by the two sides of the Presentation Layer. The most commonly used transfer syntax is the Basic Encoding Rules, BER, but it is not necessary to use BER in OSI.

The Presentation Layer calls on the Session Layer to conduct an orderly conversation with the other side, including graceful termination of the conversation and choosing of full or half-duplex communication. (Session performs many other functions, such as synchronization, activity management, and so forth, which are not detailed here.) Session conducts its conversation by means of a reliable end-to-end Transport connection, which in turn uses the Network Layer to route the data through the network. Details peculiar to the medium which is used (Token Ring, 802.3 Ethernet, etc.) are handled in the Data Link Layer, which in turn is dependent on the lowest Physical Layer for the actual electrical and physical connections between systems.

Transition from TCP/IP to OSI

Since TCP/IP is so successful, one may ask "Why bother?" when it comes to conversion to OSI. The first answer is the promise of applications which are much richer in function than those available with Internet protocols. Although only a limited number of these applications are available now, there will be many more in the future as more customers start to use OSI. The Message Handling System protocols (MHS) can send many more kinds of mail than the text-based Internet Simple Mail Transfer Protocol (SMTP) can. It defines a general-purpose third-party transfer facility, with special kinds of structured mail, such as messages with binary, voice or image parts, and Electronic Data Interchange (EDI) for exchange of financial information between enterprises. The OSI global directory service, using the X.500 standard, is already in widespread use. It enables the location of application programs all over the world on any subnet which can connect to a directory service. The new Transaction Processing (TP) standards will allow the customer's TP monitor, database, and application to be purchased from different vendors and still interoperate. The Commitment, Concurrency and Recovery (CCR) protocol provides the two-phase commit needed in TP and database manipulation. The OSI Virtual Terminal protocol potentially will allow a user to log in to any system in the world from any kind of workstation. The File Transfer, Access and Management (FTAM) services provide more than just the file transfer capability of the Internet File Transfer Protocol (FTP), such as remote database access, performing of actions on the remote files, access control, and handling of many different kinds of files.

The second reason for the broad potential of OSI is that it is composed of standards reached by formal, international agreements. They are thus politically neutral and less likely to change. OSI is in widespread use in Europe, and is often the network of choice for large, multi-vendor corporate networks.

The problem of making the transition from TCP/IP to OSI has been attacked in several different ways. See in particular Marshall Rose's book, *The Open Book* [ROSE90], which contains an entire section on *Transition to OSI.* But none of these documented approaches answer the problem of converting the X Window System to run over OSI.

One approach is the Application-Gateway. This is a program which converts analogous protocols from one suite to the other. For example, an FTAM-FTP Gateway can send files between OSI and TCP/IP systems, because it converts FTAM commands and responses into FTP commands and responses and vice-versa. The functions of the OSI FTAM must be truncated into the more limited functions of FTP. In the case of the X Window System problem at hand, the application running on both OSI and TCP/IP is **exactly the same application.** Thus no conversion of functionality is required. Although we did in fact write a Gateway, it does not need to understand the X requests that are made. It just passes the X protocol through to the other side, unexamined. A true Application-Gateway must be able to understand the protocol requests of each domain and translate them to an equivalent, or compromise, request in the other domain.

A second transition approach is to use a Transport-Gateway, Transport Service Bridge, or Network-Service Tunnel. These methods involve running the upper OSI layers on top of either TCP or IP. They are handy when you wish to experiment with OSI-based applications on systems which do not support the OSI lower layers. In our case, however, we had a TCP/IP-based application which needed conversion, and we had OSI products available on all of our systems.

A third approach, the Dual-Stack, is also used for unlike applications to talk to each other, and requires installation of both protocol suites. As with the Application-Gateway method, this method does not apply since we have one application, which we want to talk directly to the protocol stack on systems which have either OSI or TCP/IP installed.

Our approach is a fourth method. We claim that rewriting TCP/IP based applications to run over OSI can be easier than any of the approaches listed above, if a simple, high level interface to OSI is provided. The job is particularly straightforward if the high level interface is the socket interface, since many TCP/IP-based applications are written to this interface.

Standards

As shown above in the case of both X and OSI, in order for true communication to exist, standards are necessary, so independent implementations will interoperate. The X Window System is already a de facto industry standard. The definition of the protocol between X client and X server will soon be published as an American National Standard (ANS) [ANSI91], and will then become an international standard (IS). One of the authors contributed to Part IV of this draft standard, the *Mapping onto Open Systems Interconnection (OSI) Services*.

In this mapping, the X Window System applications, client and server, are found in the Application Layer of the OSI Reference Model. The Association Control Service Element (ACSE) manages the connection between client and server. The client sends the first X data, which is the Open Display request of the X protocol. This request, and all subsequent X requests, replies, events, and errors, are carried as User Data on Presentation Layer P-Data requests.

International consensus has been reached on X over OSI. The European Workshop for Open Systems (EWOS) has published the EWOS Technical Guide 013, *A Mapping of the X Window System over an OSI Stack*. This ETG specifies a minimal use of ACSE, Presentation and Session services, known colloquially as the "skinny stack," for use with X. In a *Guidance for Implementors* section, the ETG suggests that a Transport Layer API be used, and that fixed headers for the upper layers be prepended to the outgoing data and removed from incoming data. In work parallel to ours, the University of London Computer Centre, under the sponsorship of the UK's Joint Network Team, has implemented this "skinny stack" method on two systems [JNT].

While it may seem obvious that X, being an application, should operate in the Application Layer of the OSI Reference Model, this was not always the case. Initial work in mapping X to OSI placed it just above the Transport Layer, since it was felt that this layer was more closely analogous to TCP [BRENN91], [CROWC90]. This placement, however, is not a valid use of OSI. The OSI Transport Layer functions are not completely the same as TCP functions, and the OSI Reference Model assigns functions to the upper layers which are needed for valid communication. It was felt that a Transport Layer mapping would not be approved by ISO committees as an international standard since it violates the OSI Reference Model. In addition, Application Layer placement allows for future changes to the X protocol which can take advantage of the richness of OSI functionality, such as the ability to address multiple servers, OSI security aspects, and the ability to specify multiple transfer syntaxes including one for compressed data.

Implementation

In order to prove the feasibility of X over OSI, we implemented prototypes on AIX and on two other IBM operating systems - VM and OS/2 - which support the socket interface to TCP/IP by means of a user-space application library. IBM ships OSI products on all of these operating systems. On the RISC System/6000, the OSI product is called OSI Messaging and Filing, or OSIMF/6000 [OSIMF91]. For VM and OS/2, the OSI product is called OSI/Communications Subsystem, or OSI/CS [OSICS].

Because the Application Programming Interface to OSI is different in OSI/CS and OSIMF/6000, different approaches were used. This also provided the opportunity of comparing the methods for ease of implementation, minimization of changes to the X code, network management ability, and ease of conformance testing.

The goals of implementation of X over OSI across all three operating systems were the following:

- Enable X server and X client to support both OSI and TCP/IP protocol families at the same time. The reason for this goal is that many potential customers of X over OSI already have TCP/IP installed, and are adding OSI to their repertoire.
- Enable IBM X servers and X clients to support only one of the two protocol families, for systems on which only one is installed. This is done by means of conditional compilation and/or selective linking of the X code.
- Minimize changes to the X code. This was a goal even for the approach in which the changes were concentrated in the X code rather than in the socket layer.
- Maintain existing semantics of the X code. This goal means that, for example, we did not redesign the X client library to do synchronous (blocking) I/O, even though this would have made our job easier. The reason for this is that client applications are allowed to have connections to multiple servers at one time. Thus asynchronous I/O must be used.
- Make it as easy as possible for human users of X clients to specify the OSI addresses of X servers. OSI addresses are long and difficult to type. We wanted to be able to use nicknames, resolved to an OSI address by directory lookup.
- Require absolutely no changes to X clients themselves, except to specify that they must link with a particular Xlib. There is a large body of existing X clients. We did not want to add an invocation parameter to all of these.
- Make no changes to the invocation parameters for X servers, so that switching to the

enhanced X server is transparent to workstation users.

A primary goal of the design of OSI support was to allow the enhanced server and Xlib to support **both** OSI and TCP/IP at the same time. Thus, the server can accept connections from both TCP/IP-based clients and OSI-based clients, if it is running on a system on which both protocol families are installed. The client linking with the enhanced Xlib can connect to either an OSI-based server or a TCP/IP-based server if both are installed on the client host. This raised the problem of how to distinguish which protocol family is desired when the client is invoked. Currently, with TCP/IP, the server desired is specified as follows:

```
-display serveripname:x.y
```

where serveripname is the Internet host name of the system on which the server is running, x is the number of the display desired, and y is the number of the screen for that display. DECnet names are distinguished from Internet names by specifying a double colon, as follows:

```
-display serverdecnetname::x.y
```

The Xlib XOpenDisplay routine knows to attempt a DECnet connection if the server's name is specified with two colons following it.

We decided that a similar syntactical method of identifying the OSI protocol family would be beneficial. We selected the following method:

```
-display serverosiname:ix.y
```

where the *i* identifies the host name as being an OSI nickname used for OSI directory lookup. This method means that the user running the X client can select the protocol family desired, and there is no problem with duplication of Internet names with OSI nicknames. An OSI nickname is a short mnemonic name for a system or X server. This nickname is turned into a full OSI address (which may be quite lengthy and difficult to remember) by the software.

The X Window System code as distributed by the MIT X Consortium uses the BSD socket to access TCP/IP. The VM and OS/2 design consisted of modifying the existing application socket library (now shipped as part of the TCP/IP products) to invoke OSI at the ACSE/Presentation level, using the API provided by OSI/CS. Using this approach we were able to avoid changing the X code very much, and we could take advantage of the OSI upper layers in the product. Because of this approach's advantages, we also implemented an OSI socket library on AIX.

To result in an upper layer OSI socket, two sets of functional modifications were needed on the socket architecture. First, the socket architecture as implemented by IBM was redesigned to match the OSI socket support found in the 4.3 Reno release of Berkeley UNIX [BSD43]. A new family was defined to allow sockets to be declared as belonging to the OSI address family, and new address structures for this family were introduced.

Second, the 4.3 BSD socket design was extended to support OSI at the ACSE/Presentation Layer, so that it could be used by the X Window System. Three new protocol definitions were provided so that an application can specify that either Presentation layer, Session layer, or Transport layer protocols are to be used by the OSI socket being defined. New setsockopt and getsockopt services were added to specify user information, application context name, presentation context definition list, and maximum length of user information [CROWT91]. This new information is stored in the socket data structure.

Skinny Stack Implementation

On AIX, the first method tried was to modify the X server and X client library source code to support OSI by adding conditional compilation of two commonly used interfaces to the OSI Transport Layer. These two API's are the socket API, taken from the 4.3 Reno BSD OSI socket implementation [BSD43], and the X/Open Transport Interface(XTI) [XOPEN88], which is an important new standard for network programming and is the one used in IBM's OSIMF/6000 product.

The goal of this preliminary AIX work was to revise the Xlib and the sample X server code to support the XTI interface to either OSI or TCP/IP, and to support OSI with either the XTI or socket interface. Since X is written in a very clean, modular manner, the changes were confined to a very few existing X routines: four in Xlib and three in the X server, plus a new module for each of client and server. A new module for XTI support, and a module for OSI support, linked by both client and server, were added. A new module for OSI directory support, useable by any OSI-based application on AIX, was written. Since no OSI directory services are available with OSIMF, these routines needed to be written.

Because the server and client library must be able to support OSI, TCP/IP and UNIX-domain connections, a new data structure used by all connections had to be added. This data structure specifies the protocol family and endpoint type (socket or XTI). The *connection.c* module of the server contains routines for each protocol family supported which make the connection. A minor change had to be made to each of these to set up this new data structure. In both client and server, the data structure is tested on each read or write to determine which API to use (socket or XTI) and which domain to use (OSI, UNIX, or TCP/IP).

The OSI and XTI support was added with the following new or modified C language statements: Xlib, 250 statements; X server, 250 statements; new code used in both client and server, 1000 statements[1].

The work for the prototype was done on the brand-new Release 5 of the X11 source code. When an attempt was made to retrofit the X changes to the product release of AIXWindows, which at the time the work was done was Release 3 of X11, the overriding disadvantages of using this approach became clear. It was difficult to do the retrofit work for the server because key modules had been changed by MIT. The prospect of repeating this work for each future release of X was unpleasant. In addition, we had invested a lot of time in this work and the result was only that the X Window System now worked over OSI, but not any other application.

Because of these two problems, we decided to rewrite the code into a user-space socket emulation library. In AIX, sockets are of course part of the kernel. We wanted to get this implementation working quickly, without the complications of kernel modification. AIX does not currently support socket access to OSI at any layer. Our socket emulation library, thus, intercepts socket calls in user-space, determines whether this is a real socket or emulated socket, and makes the real socket call if requested. The emulated socket code issues the XTI call, and translates the return codes to socket return codes before returning to the caller. Much of the code from the embedded method was used intact in the socket library. The per-connection data structure was simply turned into the emulated socket data structure, set up on every call to the new Socket service (even for TCP/IP and UNIX domain sockets). The XTI access routines are now called by the socket emulation library rather than called by X routines.

Since the X code was already written to the socket interface, changes to use this new socket emulation library were minimal. Thus retrofitting these changes to future releases of X will be painless.

The number of lines of code for the changes done by this method was about the same as the previous method, but the number of lines modified in the actual X code was substantially less – under 100 for both client library and server. The bulk of the new code is in separate modules implementing the OSI socket library.

[1]All counts of statements in this paper were arrived at by counting lines of code ending in a semi-colon. This includes data declarations, and does not include lines consisting only of comments, curly brackets, preprocessor statements, or *if (expression)*.

In both implementation techniques, the upper OSI layers were needed, since in OSIMF/6000 the only OSI API is to the Transport Layer. This required implementation of specialized ACSE, Presentation, and Session Layers, thus lending itself to use of the "skinny stack" method envisioned in the EWOS Technical Guide. That is, the layers implement only the simple functions needed by the X Window System. On outgoing data, predetermined fixed headers for the upper layers are prepended to the X data itself, and passed out over the Transport layer. Incoming data is more complicated; it must be parsed to determine the header type and format of the data. The headers are stripped from the data, and only the X data itself is passed up to the caller.

The advantage to this approach is that the OSI Layer implementations were hand-coded for use by TCP/IP-based applications such as X only. Thus tests for the many unused functions of the Session Layer, for example, were eliminated. The Presentation Layer implementation does not need to handle encoding of general abstract syntaxes. In the socket approach, these upper OSI layers are embedded in the socket emulation library, below the socket API.

The initial implementation contains no network management in the upper layers, a disadvantage to this approach. If we had been able to use a complete, general-purpose OSI implementation, network management would have been included. Another disadvantage to this approach is that the specialized upper layer code must now be tested for conformance with standard OSI layers. While the EWOS Technical Guide restrictions permit simplified encoding of outgoing data, all possible forms of incoming data must be decoded, and these must all be tested. If we had been able to use a product level implementation of the OSI upper layers, this testing would have already been done.

A rule of thumb for performance of the X Window System is that the round-trip time for communication between client and server should be 50 milliseconds or less in order for human perception of response time to be acceptable. Performance of the OSI-based X server and X client library was measured by using *x11perf*, which is an X client used for measuring server performance. This client calculates and displays the round-trip time of communication between client and server before going on to execute detailed tests of the server graphics operations. This round trip time was found to be approximately 20 ms, with untuned code, which is an acceptable level. This number was dominated by the performance of the OSIMF lower layers. Running at Transport layer versus running at ACSE layer made very little difference in the round-trip time.

General Purpose OSI Product Implementation

For both VM and OS/2, IBM ships a socket library as part of their TCP/IP products. We obtained the source code for these libraries and modified it to support OSI as well as TCP/IP.

On VM, only X client applications are supported. The availability of X over OSI means that a VM customer running only the OSI communications protocol on their machine can execute X-based applications on their VM mainframe and take advantage of the display capabilities of a bitmap graphical terminal attached to an X server connected to an OSI network.

The goal of the design for VM was to provide the ability for any program using the socket interface for network communications to use TCP/IP, OSI and local sockets simultaneously. In the case of X, this permits a single client application to communicate with multiple remote X servers, some of which might be using TCP/IP transport protocol and others using OSI transport protocols. The socket application library was modified to support the OSI/CS ACSE/Presentation Layer API. This means that the socket library need not contain an implementation of the OSI upper layers, as was necessary in AIX.

The TCP/IP socket code was modified to test for the domain address family on each request. For most services, new routines were written for the OSI protocols and added to the socket library; in a few instances existing routines were simply modified to use the OSI/CS API to invoke the appropriate ACSE/Presentation Layer function. A major benefit of this approach is that it will allow any application (not just the X Window System!) which utilizes the socket for network communication to replace TCP/IP with OSI with only minor changes required to handle specification of the OSI protocol and address family. As a result, aside from the socket library changes, only minor changes to the X library were made to set up information required for OSI connections.

The OSI implementation used is an IBM product, and contains network management in all layers, as well as extensive directory services. It allows the clients and server to refer to each other very simply by nickname, so that there was no need to write any routines to perform directory lookup of OSI addresses. In addition, the OSI implementation is already conformance tested. Thus testing required for this work could be confined to the socket changes and X changes only.

The OSI support was provided by adding or modifying 100 C statements in Xlib, and 1600 C statements of socket library code to utilize the API provided by OSI/CS. Note that this is about the same effort as the AIX X code modifications, even though the AIX code is implementing the upper layers themselves, and the VM code is using an API to the upper layers. On first examination the number of lines of code that was added to the socket library to provide these services might seem surprising. However, an examination of the code reveals that there are three factors that account for this size. First, the socket library changes represent a general purpose implementation of services that are not optimized for usage by X. Modifications were made to provide consistency with the framework of the existing socket library structure. Two, because X code handles multiple connections at once, for this implementation most OSI/CS callable services are performed asynchronously and control is returned to the issuing socket routine as soon as the callable service is received by OSI/CS, but before the action is complete. Return indicates only that the parameters of the OSI service have undergone a preliminary stage of validation, and that the call has been accepted (or rejected) for further processing. This means that for each asynchronous OSI/CS service call that was used, an additional test had to be added to the socket code to determine the results of this preliminary validation. And finally, all return codes and error conditions that are received from OSI/CS must be converted to corresponding socket return codes and error conditions. Due to the large number of possible codes and conditions that can be returned from OSI/CS, a significant portion of the new code represents tests that are performed in the event that an error condition is received, and the subsequent conversion of these codes and conditions to socket codes and error numbers.

On OS/2, IBM currently ships an X server, called PMX, and no X client library. Availability of an OSI-based X server means that clients running on an OSI-based system can display on the PS/2 workstation. X server requests that are received from an X client program are interpreted and OS/2 Presentation Manager (PM) requests are used to control the workstation's bitmap display. Keyboard and mouse events, error notifications, and request replies from the server are packaged in X protocol packets and transported over the network to the client.

Since IBM provides OSI support for OS/2 with its OSI/CS product, we originally planned to implement the X/OSI support using an approach similar to that used in the VM implementation; we would modify the socket code allowing application programs to remain largely unchanged. TCP/IP socket source code for OS/2 V2 was obtained, and the implementation designed using this model. However, pending the availability of a OS/2 V2.0 Ethernet device driver for OSI/CS, it was necessary to change the design to use an approach similar to the first AIX implementation – modifications and additions were made to the PMX server. The OS/2 X server code code performs all tests for protocol family determination, and invokes the calls to the OSI/CS API as required.

As in the first AIX implementation, since the PMX server code has been modified this current OS/2 implementation will require that the changes be retrofitted whenever there are changes made to the PMX code on which it is based. Fortunately, modifications to existing server code was confined to two modules with 100 new C statements added. All additional changes were implemented in three new modules, representing an additional 1700 C statements. As in the case of VM, these new modules provide general purpose OSI functional equivalents of socket service requests, and are available to any application. However, since these changes were not made to the socket library (due to the device driver availability problem previously mentioned) applications must explicitly call for the OSI version of requests (e.g., osi_socket, osi_listen, osi_read, etc.).

Finally, as in the case of the VM implementation and in contrast to the AIX implementation, the OSI support is provided by the IBM OSI/CS product, and therefore a complete OSI implementation including network management, extensive directory services, as well as conformance testing are all provided by this product.

X TCP-OSI Gateway

A typical customer who is beginning to install OSI will have some systems running OSI, but some which are still running TCP/IP. The customer will want to use the X Window System to communicate among all of these systems. To satisfy this need we implemented the X TCP-OSI Gateway. This program reads data on one protocol family and writes it out on the other, thus connecting clients and servers running different protocol families. It runs on AIX and VM.

Figure 3 shows a diagram of a mixed network with some systems running TCP/IP and some running OSI. Through the intermediary of the X TCP-OSI Gateway, these systems can talk to each other.

The Gateway appears as an X server to normal X clients, and appears as an X client to normal X servers. It must be able to interpret addresses in both OSI and Internet address domains. It runs on a system which has both OSI and TCP/IP installed. The Gateway was written by adapting some of the network communication routines found in the server, and linking with Xlib to use the network communication routines found in the client. The Gateway does no interpretation of X protocol requests or replies. It simply passes them uninterpreted to the other side. Because of the specific data exchange made when an X client connects to an X server, this Gateway will work only for X. However similar work could be done for other distributed applications.

Future Work

Now that we have OSI socket libraries we plan to port other socket-based applications to OSI. Candidates for this work include TELNET, FTP, NFS, x3270. This will enable users to maintain their

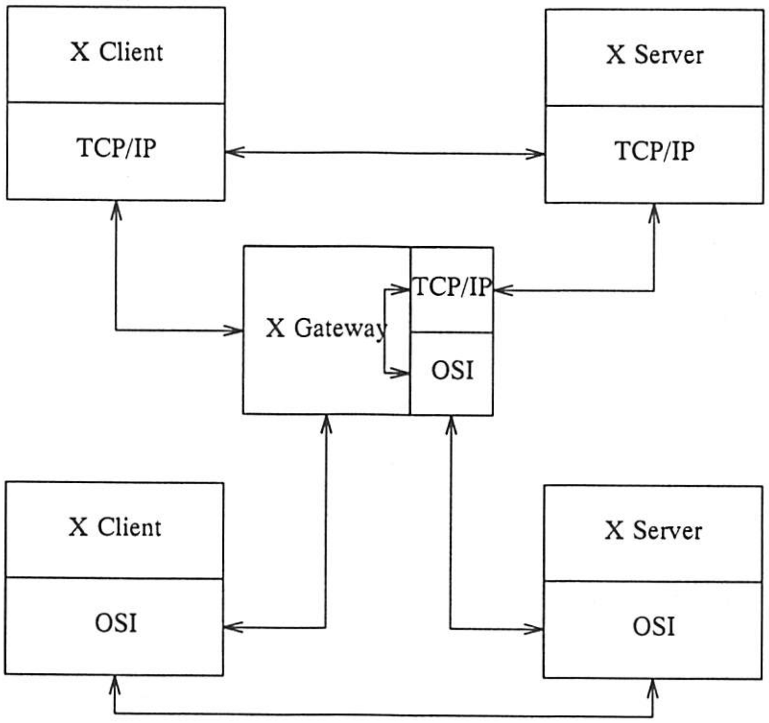


Figure 3: Use of X TCP-OSI Gateway

productivity with familiar network tools, while adding new OSI applications to their repertoire.

Conclusions

The problem of moving a TCP/IP-based application such as the X Window System to OSI was addressed. Different approaches were taken in different operating systems and these were compared. The "skinny stack" approach was used in AIX. Specialized, minimal function OSI layers were implemented by prepending the OSI layer headers to data sent using a Transport Layer API. This approach minimizes the path-length required for OSI, but means that we could not take advantage of upper layer network management or conformance testing.

When the "skinny stack" code was embedded in the X source code, retrofitting problems arose. On systems where the socket interface is found in a user-space application library, the socket was modified to access OSI at the Application Layer. Modification of an existing socket interface to support OSI means that the changes to existing X Window System product code can be confined to under 100 lines of code. This method proved to be so superior in terms of minimization of changes to the X source code and usefulness for other applications, that a user-space "socket" library was implemented on a UNIX system where sockets are found in the kernel. Proper implementation of an OSI upper layer socket clearly requires kernel changes. Once the socket modifications are made and tested, they can then be used by any application written to the socket interface. The socket implementation can use an upper layer access to OSI if this is available to kernel code (often it is not), or can implement the "skinny stack".

Although direct performance comparisons between the "skinny stack" method and general purpose OSI layer method could not be made since both methods could not be implemented on the same operating system, one would suspect that a "skinny stack" would exhibit higher performance. However, a general purpose stack can be implemented in such a way that a fast path is used for applications which do not use the complicated features.

For operating systems which are slow to implement OSI, a Gateway can be interposed between clients and servers in different protocol domains. This Gateway is particularly easy to write if a high-level interface such as the socket is available for both domains.

Based on our experience, we advocate that common network API's such as sockets, XTI, and TLI, should be modified to support OSI at the Application Layer as well as the Transport Layer. The OSI upper layers should contain network management and directory support, and should be conformance tested. The simplifications and fast-path approach highlighted by the "skinny stack" should be used as much as possible. When these high level interfaces are available in operating systems, existing TCP/IP-based application which are now in widespread use can be rewritten with only minor changes to run over OSI.

Although these experimental prototypes are not available to customers, they point the way toward solution of the problem of migrating all TCP/IP-based applications to OSI. An OSI upper layer socket, which implements the most basic, simplified functions of Session, Presentation, and ACSE, can be used by all socket-based TCP/IP applications. Since the more complicated functions of the OSI upper layers were not available to them, they either do not use them, or implement them in other ways. This capability is critical for users who want to change to OSI in order to comply to the Government OSI Profile, or who want to start using the rich functionality of OSI applications, or who want to communicate with European OSI-based networks, but do not want to give up using their TCP/IP applications.

Acknowledgements

We wish to thank our management at the IBM Cambridge Scientific Center, Bob Anderson and Dick MacKinnon, for funding and encouraging this work. Jerry Mouton of IBM Networking Systems had the original idea for the OSI upper layer socket, and both he and Keith Sklower of the University of California made contributions to the design of this socket. We would like to thank Jay Elinsky and Oleg Vishnepolsky of IBM Watson Research, and the PMX team at Cambridge – Allen Springer, Bill Barrett, and Rod Maxwell – for making their socket library source code and PMX source code available to us.

Bibliography

[ANSI91] X3.196-199x, X Window System Data Stream Definition. Part I, *Functional Specification*. Part II, *Data Stream Encoding*. Part III, *KEYSYM Encoding*. Part IV, *Mapping onto Open Systems Interconnection (OSI) Services*. Revised November 20, 1991.

[BRENN91] Brennan, Thompson, and Wilder, *Mapping the X Window onto Open Systems Interconnection Standards*, IEEE Network Magazine, May 1991.

[CROWT91] Crowther, *X on OSI*, Xhibition 91 Conference Proceedings, Integrated Computer Solutions, June 1991.

[BSD43] Manual pages for socket calls in 4.3 Reno release of Berkeley Software Distribution, Sections 2 and 4, Computer Systems Research Group, Computer Science Division, Univ. of California, Berkeley, Calif, May 30, 1990.

[CROWC90] Crowcroft, J., *Experience with mapping the X windows protocol onto ISO transport* *service,"*, Networks 90 - Network Management. Proceedings of the International Conference, Birmingham, UK June 1990

[DYER90] Dyer,"X Windows over OSI", Report of Joint Network Team, Rutherford Appleton Laboratory, United Kingdom, October 1990.

[EWOS91] EWOS Technical Guide 013, *A Mapping of the X Window System Over an OSI Stack*, European Workshop for Open Systems, 21 May 1991.

[GOSIP88] U.S. Government Open Systems Interconnection Profile (GOSIP), U.S. Federal Information Processing Standards Publication (FIPS) 146, Version 1, August 1988.

[IS7498] ISO 7498, Information Processing Systems – Open Systems Interconnection – Basic Reference Model, International Organization for Standardization, Geneva, Switzerland (1983).

[IS8823] ISO 8823, Information Processing Systems – Open Systems Interconnection – Connection oriented presentation protocol definition, Geneva, Switzerland (1988).

[JNT] Peter Furniss and Kevin Ashley, personal communications with the authors.

[OSICS] *OSI/Communications Subsystem General Information Manual*, GL23-0184, IBM, Palo Alto, California, March 1990.

[OSIMF91] AIX Version 3 for RISC System/6000 OSI Messaging and Filing/6000, User's and System Administrator's Guide, Second Edition, SC32-0012-01, IBM, Palo Alto, California, September 1991.

[ROSE90] Rose, *The Open Book*, Prentice Hall, 1990.

[SCHE90] Scheifler & Gettys, *X Window System*, Second Edition, Digital Press, 1990.

[XOPEN88] X/Open Portability Guide 3, Volume 7 Networking Services, X/Open Company, Ltd., Reading, Berkshire, United Kingdom, 1988.

Author Information

Nancy Crowther has been at IBM for 15 years. Most recently she has been a Staff Member at the IBM Cambridge Scientific Center, where she has worked in the area of networking software. She also worked for Informatics at NASA/Ames Research Center where she did scientific applications for 6 years. She has an M.S. in Applied Mathematics from the University of Santa Clara and an A.B. in Mathematics from Rutgers. Reaçh her via US Mail at IBM, 101 Main St., Cambridge, MA 02142, or electronically at crowther@cambridge.ibm.com.

Joyce Graham is a Staff Member at the IBM Cambridge Scientific Center. Prior to joining IBM, she worked at United Technologies Pratt & Whitney Division as a research engineer, and at the Massachusetts Institute of Technology as a scientific programmer, user consultant, and systems programmer. Since joining IBM in 1981, she has worked in the areas of operating systems, user interface, and network communications. Ms. Graham received a joint B.A./B.S. (Aeronautical Engineering) from New York University. She may be reached at joyce@cambridge.ibm.com or via US Mail at IBM, 101 Main St., Cambridge, MA 02142.

ISBN 1-880446-44-8